Interactive Ray Tracing of Massive and Deformable Models

Christian Lauterbach

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2010

Approved by:

Dinesh Manocha, Advisor

Anselmo Lastra, Reader

Ming Lin, Reader

David Luebke, Reader

Jan F. Prins, Reader

Abstract

Christian Lauterbach: Interactive Ray Tracing of Massive and Deformable Models.
(Under the direction of Dinesh Manocha.)

Ray tracing is a fundamental algorithm used for many applications such as computer graphics, geometric simulation, collision detection and line-of-sight computation. Even though the performance of ray tracing algorithms scales with the model complexity, the high memory requirements and the use of static hierarchical structures pose problems with massive models and dynamic data-sets. We present several approaches to address these problems based on new acceleration structures and traversal algorithms. We introduce a compact representation for storing the model and hierarchy while ray tracing triangle meshes that can reduce the memory footprint by up to 80%, while maintaining high performance. As a result, can ray trace massive models with hundreds of millions of triangles on workstations with a few gigabytes of memory. We also show how to use bounding volume hierarchies for ray tracing complex models with interactive performance. In order to handle dynamic scenes, we use refitting algorithms and also present highly-parallel GPU-based algorithms to reconstruct the hierarchies. In practice, our method can construct hierarchies for models with hundreds of thousands of triangles at interactive speeds. Finally, we demonstrate several applications that are enabled by these algorithms. Using deformable BVH and fast data parallel techniques, we introduce a geometric sound propagation algorithm that can run on complex deformable scenes interactively and orders of magnitude faster than comparable previous approaches. In addition, we also use these hierarchical algorithms for fast collision detection between deformable models and GPU rendering of shadows on massive models by employing our compact representations for hybrid ray tracing and rasterization.

Acknowledgments

I owe a debt of gratitude to the many people who made this thesis possible. First and foremost I would like to thank my advisor Dinesh Manocha who over the years has always been supportive, encouraging and patient when necessary. His influence has shaped and guided much of the research in this thesis, and his experience and advice have been invaluable many times. I am equally grateful to Ming Lin and all my co-workers in UNC's GAMMA group, many of which have become friends and co-authors over the years. I also would like to thank everybody in Intel's Advanced Graphics Lab (now Intel Labs) as well as NVIDIA Research. The summers spent as an intern have been an enormous inspiration to my research; many of the ideas in this thesis might not have been conceived without this time and everybody. Of course, I am also indebted to my thesis committee, Dave Luebke, Anselmo Lastra and Jan Prins whose feedback and perspective have made this a better thesis than it would have otherwise been.

Last, and certainy not least, I would like to thank my family for all the encouragement and support of my research (even though it meant moving half-way around the world) and of course my wife Sarah for her love and her tolerance of the many late nights and missed weekends in the name of paper deadlines. I dedicate this thesis to her.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Ray Tracing

The rules of geometrical optics have been used for centuries to compute the path of light as straight rays in physics or engineering. As a computer algorithm, ray tracing [6] is a fundamental technique used in many areas of computer science. In computer graphics, it simulates the propagation of light, e.g. as rays from the eye [158, 78] or photons from light emitters [73]. Thus, it can be used to synthesize realistic images. In other applications, geometric simulation algorithms use rays as a representation of other waves or particles and can perform ray tracing to simulate other phenomena. For example, acoustical algorithms approximate the path of sound waves by stochastically sampling rays [89] and computing paths to simulate room reverberation. Similarly, radio waves can be traced such as to estimate signal strength based on emitters in a virtual scene. In nuclear simulations, rays represent the trajectory of particles and can be used to compute fission reactions. As a query to test whether two points can visible to each other, ray tracing is also a useful tool in other areas. For example, in collision detection rays can be shot to estimate whether there are obstacles ahead. In line-of-sight computations such as in training systems, rays can determine whether objects are visible to each other. While this thesis primarily deals with ray tracing in the context of

Computer Graphics, the methods described here are equally applicable in other areas.

Given a ray and a scene described mathematically, e.g. as a set of geometric primitives such as triangles, the basic operation in ray tracing is to find the intersection of that ray with the scene, if any. The trivial way to do so is to intersect the ray with every primitive and then return the closest (or any) intersection. For obvious reasons, this quickly becomes inefficient when the scene consists of more than a handful of individual primitives. Acceleration structures such as grids and hierarchies for ray tracing have the purpose of allowing to quickly determine which objects a given ray should be intersected with to get the closest intersections, and to do so in less than linear time in terms of the number of primitives.

While many such acceleration structures have been proposed, in the context of this thesis the most interesting ones are hierarchies such as kd-trees or bounding volume hierarchies (BVHs) since they are most universally adaptive to most kinds of scenes [61]. Hierarchical acceleration structures are a representation of objects as a tree where each node represents a subset of the objects, starting with the root node for all objects and down to the leafs with typically only one of very few objects. Each node also provides a simple geometric shape (e.g. a box or sphere) that the ray is tested for intersection against, with the special property that if the ray hits any object contained in the node, it must also intersect the boundary of the node. This allows us to quickly cull away large non-intersecting parts of the scene objects during ray tracing. The use of hierarchies in ray tracing is thus a recursive search traversal of the tree structure. For n objects and assuming that the depth of the tree is on average $O(\log n)$, tracing a ray through the hierarchy therefore has an average run-time of $O(\log n)$ as well, compared to $O(n)$ for testing each object.

In Computer Graphics, the recent focus in ray tracing has been on improving the performance of ray tracing to allow interactive performance as a rendering technique and as an alternative to hardware-accelerated rasterization rendering that has proven

less flexible and general in terms of which aspects of light transport it can simulate. Some of the recent methods include bundling rays into small packets that can be traced together and evaluated using vector operations [151] on commodity processors. Another recent trend has been to develop improved acceleration structures [100, 61] to perform ray queries. For the latter, it has been shown that the highest performance is usually attained with acceleration structures that perform a very detailed and fine-grained subdivision of the objects and thus provide the smallest set of primitives to intersect with each ray. In other words, it is more efficient to spend more time traversing the hierarchy in order to save time in performing ray intersection tests with the primitives. However, when handling massive models with tens or hundreds of millions of triangles this leads to other problems, as described in the next section.

1.2 Rendering complex models

The complexity of geometric models, measured in terms of number of triangles, has been steadily increasing in various applications. For many interactive applications (e.g. games or virtual reality), this has been driven by the fast evolution of graphics rasterization hardware that is able to render increasing numbers of triangles. However, for the largest of models the determining factor has been the source of the data: for example, new scanning techniques for three-dimensional objects produce a number of geometric primitives proportional to the desired accuracy, which can often exceed the capabilities of the rasterization hardware. CAD applications for complex machinery or architectural objects have high level of detail for most objects since they are modeled with manufacturing or physical construction in mind. The combination of all the parts in the model (e.g. the CAD model of a Boeing 777 airplane as shown in Fig. 1.1) often creates overall geometric complexity that far exceeds the rasterization capabilities available. Models originating from large scientific simulations have a complexity that is dependent on the

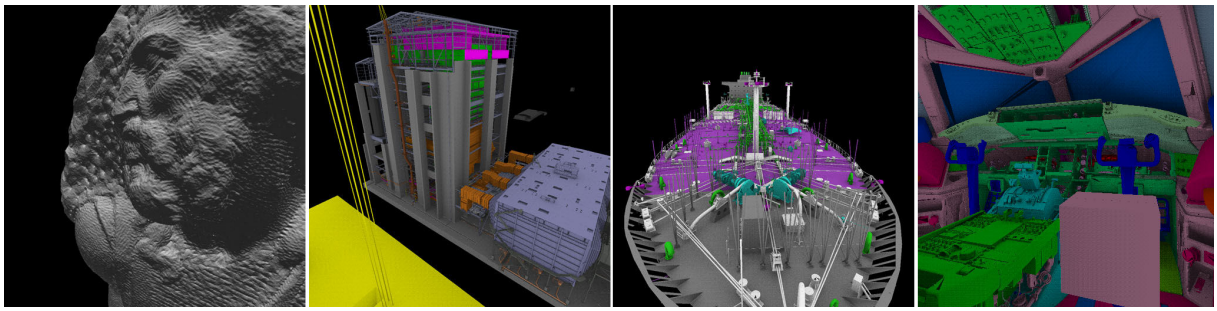


Figure 1.1: **Complex models:** *Several highly complex models: From left to right: St.Matthew (372M tris), Power plant (12.7M tris), Double Eagle tanker (82M tris), Boeing 777 (360M tris). The right three models are examples of CAD and architectural models, whereas the leftmost one is the result of laser-scanning a three-dimensional object.*

desired accuracy in the respective domain, not the visualization tool. Often, the resulting data-sets such as iso-surfaces are extremely large and hard to visualize at interactive rates. Thus, interactive visualizations of these models are often limited to only a partial or simplified view of the complete system. Providing a complete interactive rendering system for these models is very desirable for many applications such as visualization, maintenance, training or engineering.

Many approaches for rendering these kinds of models using rasterization with graphics hardware acceleration to allow interactive walkthroughs have been developed. Since the rasterization paradigm draws all the triangles, this has linear time complexity and naive rendering of these models is unlikely to be fast. Some methods reduce image quality for speed, e.g. by using several different levels of detail. Here, it is assumed that simplified version of the objects can be used at a distance without a large visual difference, thus reducing the rendering workload. Others concentrate on model compression to reduce overall memory footprint, culling of non-visible or occluded objects to reduce the number of primitives that need to be rendered, or alternative rendering primitives to reduce the work per object.

An additional implication of the detail in complex models is that advanced lighting and visualization methods are needed to help the human observer understand spatial

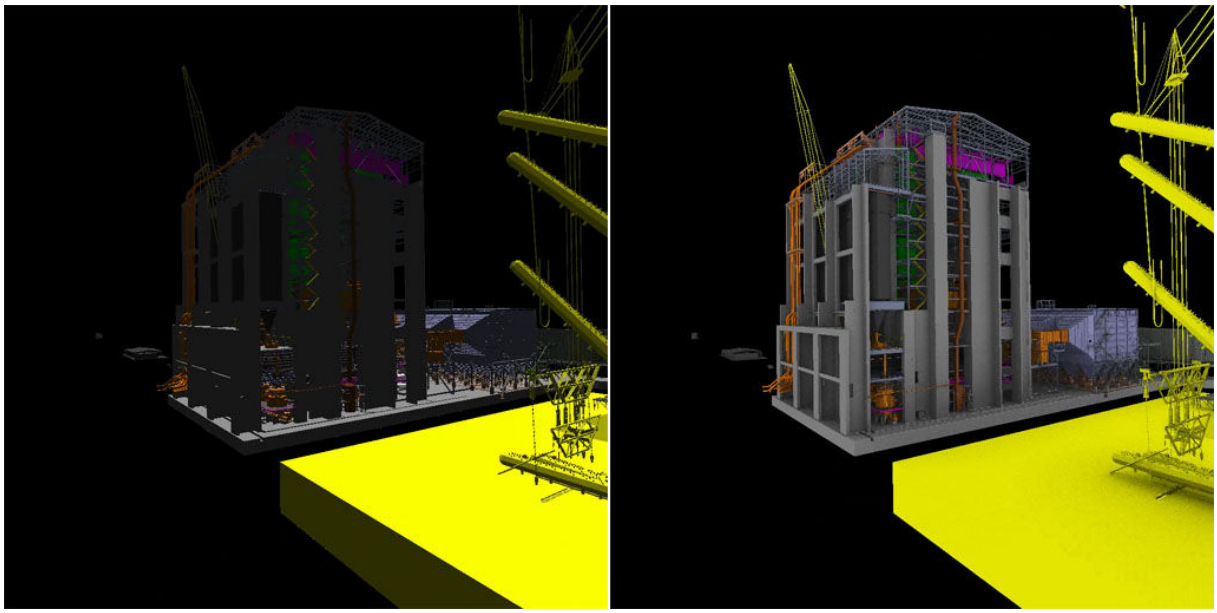


Figure 1.2: **Advanced visualization of complex models:** *The 12.7M triangle Powerplant model rendered using only local lighting (left) and with ambient occlusion (right). Using correct lighting helps with intuitive understanding of spatial relationships especially in geometrically complex models.*

relationships between the objects. Examples of this are shadows, partial transparency to see hidden objects, and indirect illumination. However, since the rendering techniques mentioned above are all based on the rasterization pipeline, it is much harder to extend them to handle secondary effects such as reflections and refractions, or non-local lighting such as soft shadows. Other effects such as arbitrary transparency also pose hard problems in complex models. Therefore, while it is possible to render massive models relatively fast using rasterization, it is hard to generated realistic images.

1.3 Issues in ray tracing of complex models

Since ray tracing generally has logarithmic time complexity with the number of primitives (e.g. triangles) in the scene, it is a good choice to use on complex scenes because of its asymptotic performance. In addition, it easily extends to arbitrary lighting and visualization methods and is therefore able to support much more general, high-quality

visualization methods. One large benefit is that implementing ray queries in parallel is trivial both using data parallelism and multi-threading such as multiple cores on commodity processors. Even though no dedicated ray tracing hardware is available – unlike commodity graphics processors (GPUs) that are optimized for rasterization – the progress in interactive ray tracing techniques as well as development in parallel architectures such as multi-core CPUs and programmable many-core GPUs has allowed ray tracing to reach the performance necessary for real-time rendering [156, 2].

However, for complex models two main issues remain: 1) large memory footprint and the size of the working set, and 2) acceleration structure construction and maintenance for dynamic scenes.

Memory footprint: Interactive ray tracing needs far more memory than comparable rasterization techniques due to two reasons. First, as described above, a detailed and complex acceleration structure is needed to achieve high performance for individual ray queries. As practice has shown, hierarchies can potentially take up memory comparable in size of the geometric primitives, i.e. doubling the overall memory needed [149]. In addition, any object in the scene can be intersected with a ray at any time. This means that all geometric data for the scene has to be stored in a format that is both randomly accessible and is efficient for the intersection computation, which may need additional information such as material properties and normals. As a result, the working set of an interactive ray tracer can potentially be very large at any time. In contrast, rasterization-based algorithms typically stream the geometric data in a bulk synchronous manner and each object just has to be loaded once. This allows efficient encoding of objects in a sequential order, e.g. triangle strips for mesh objects, as well as compressed representations that are decoded during rendering. This reduces the footprint size and typically only needs a very small working set.

The problem of high memory footprint is exacerbated by the fact that ray tracing

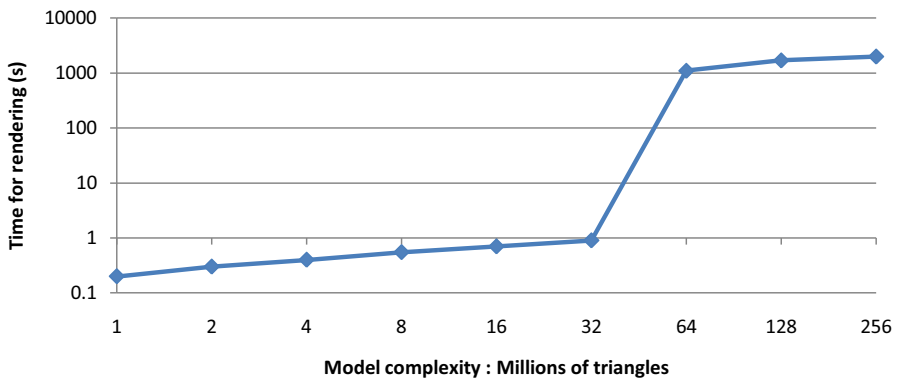


Figure 1.3: **Out-of-core ray tracing:** *The same model at different simplification levels is ray traced on a machine with limited memory (note the log-log scale). As the memory footprint of the model and hierarchy exceeds main memory beyond 32M triangles, the speed decreases by three order of magnitude due to disk cache misses. Graph from [164].*

does not perform well when the geometric data and the hierarchical data structure do not fit into the memory, i.e. during out-of-core rendering. As shown by an experiment in [164] (see Fig. 1.3), ray tracing a model that exceeds the main memory can cause the algorithm to experience enough disk cache misses that the overall performance decreases by three orders of magnitude. Even though the performance remains logarithmic with higher constants, it becomes far too slow to be of any use for interactive applications. On architectures that have severely limited memory and may not support virtual memory such as programmable GPUs, rendering out-of-core is not even a practical option and therefore the complexity of the models that can be ray traced is severely limited by the size of the available main memory.

Several techniques have been proposed to address this problem that allow ray tracing on massive models even on machines with limited memory. One approach is to change the order in which rays are evaluated so as to increase coherence in terms of memory accesses. These reordering approaches [115, 28, 36, 15] can drastically reduce the amount of cache misses and may only require parts of the model to reside in main memory, but typically only work efficiently for offline rendering with high numbers of rays. It is also possible to integrate levels-of-detail into ray tracing [20, 164, 31]. In these techniques,

precomputed simplified versions of the objects in the scene can be used during ray tracing which drastically reduces the working set size and allows to maintain the rendering performance even when the geometry footprint is larger than main memory. Another approach [153] hides memory latency for objects that are not in the main memory by substituting a simple placeholder object for those parts of the scene. However, LOD techniques may introduce visual artifacts and will also increase preprocessing time and overall memory footprint.

Finally, several algorithms have been proposed to use more compact or compressed representations for the ray tracing acceleration structures. Reduced precision BVHs [101] encode the bounding box coordinates using a quantized representation using less bits, thus drastically reducing the overall memory needed to store the BVH. However, there is significant ray tracing performance overhead due to more ray-box intersection since the quantized boxes are larger than the original ones. In addition, the decompression needed during the traversal introduces additional overhead. Another technique [21] uses higher branching factors to reduce the number of nodes and a fixed tree layout such that no child pointers are needed, but has similar drawbacks. Neither of these approaches reduces the overhead of storing the geometric data. A more general technique [82] compresses small sub-trees individually and then decompresses them at run-time when accessed. However, the run-time overhead may make this unsuitable for interactive rendering.

Hierarchy maintenance: Constructing ray tracing hierarchies is a non-trivial task that can be computationally expensive, especially when using methods that optimize the hierarchy for high ray tracing performance [61]. For complex models, computing the hierarchy in advance is possible but may add more time than the user is prepared to wait. For example, in engineering applications many virtual parts may be combined into a large model relatively quickly, but the precomputation time for ray tracing may

delay the actual rendering for minutes or hours. More importantly, for complex scenes that are dynamic – i.e. objects move or deform – the hierarchy becomes invalid as soon as the underlying geometric shape changes or the objects in the scene undergo motion. Without a fast way to construct or update the existing hierarchy, ray tracing of dynamic scenes is limited by the speed of the algorithm used to update of maintain the acceleration structure. This is ever more important on current commodity processors, as parallelism in hardware architecture increases since hierarchy maintenance does not parallelize easily and thus becomes the bottleneck in terms of scalability. Therefore, in order to perform scalable ray tracing of deformable models, it is necessary to design parallel algorithms for hierarchy maintenance and construction. In particular, current high-performance architectures such as many-core GPUs exhibit both high data parallelism in the vector units, but also have many individual cores providing high thread parallelism. To efficiently utilize these processors, it is necessary to exploit both of these sources of parallelism.

While there are approaches that can compute acceleration structures that are valid for an animation sequence [49, 57], they need a priori knowledge of the whole animation to work and typically also decrease rendering performance due to larger bounding boxes. There are two different approaches for ray tracing of animated scenes: the first is to fully rebuild the acceleration structure for each frame and the second is to refit or update the structure in order to restore correctness for geometric objects that have moved since the last frame. Several approaches have been proposed to improve hierarchy construction performance through algorithmic improvements or parallelism on multi-core CPUs (see chapter 2 for an overview), but have not been investigated for highly-parallel architectures such as GPUs.

1.4 Thesis goals

The goal of this thesis is to address these two issues in ray tracing of complex models. In particular, we want to design compact representations for interactive ray tracing of triangular models that reduce the memory footprint without degrading the overall performance significantly in terms of image quality and the frame rates. Our replace is to perform ray tracing of massive models even on commodity PC systems with limited memory such as workstation machines or graphics processors. We also seek to perform ray tracing of dynamic scenes at interactive rates such that the hierarchy maintenance between frames should not be the bottleneck of the overall rendering algorithm. We want to design algorithms that take advantage of the nature of deformable models by utilizing the coherence between frames to reduce the time spent on hierarchies. For general dynamic models, we aim to introduce methods for fast hierarchy construction on graphics processors and other massively parallel machines that fully utilize thread and data parallelism and in practice allows interactive ray tracing while reconstructing the hierarchy during each frame.

We also seek to investigate the impact of these improvements in areas other than real-time rendering. We show that hierarchy computation for dynamic models can also be used for interactive geometric sound propagation in virtual, dynamic scenes. In addition, our algorithms for parallel hierarchy maintenance also enable highly parallel collision detection. Finally, we show that rasterization and ray tracing of massive models can be combined on graphics processors for generating high-quality hard and soft shadows.

1.5 Thesis statement

It is possible to perform interactive ray tracing of massive static models and complex deformable models on commodity CPU and GPU hardware with limited memory using compact representations and highly parallel hierarchy operations.

1.6 Thesis overview and contributions

This thesis introduces several algorithms for ray tracing to address the problems listed above. We summarize the main contributions and results of the thesis here.

1.6.1 Fast, compact representations for interactive ray tracing

We propose a novel light-weight representation for triangular meshes and optimized ray tracing acceleration structures. In particular, we show how to efficiently represent geometric models based on triangle strips and object hierarchies in a structure that still provides random access for ray tracing and does incur only very small ray tracing overhead but results in a significant reduction in memory footprint compared to a standard minimal representation for interactive ray tracing. We also present a new algorithm for generating a new type of triangle strips that are optimized for ray tracing performance compared to traditional triangle strip generation algorithms in that are designed for rasterization.

Our results show that many of the widely-used complex models with several hundreds of millions of triangles can be represented in 8GB of main memory. Compared to a minimal geometry representation and state-of-the-art hierarchy, our novel ReduceM approach reduces the overall memory footprint by up to 80%. This decreases the memory complexity to a level that allows interactive ray tracing of these models on standard workstation machines as opposed to supercomputer or clusters of workstations. At the same time, using our representation results in only a very small performance overhead compared to efficient kd-tree based ray tracing [149] when both approaches run fully in-core. In terms of frame rate, at worst we observe a slowdown of 20%, but most models see almost equal performance or even an improvement in overall render times due to the smaller working set size. We also show that our algorithm for constructing triangle strips optimized for ray tracing performance has a large impact on overall speed

of ray tracing. In comparison to a standard strip construction algorithm we observe a performance increase of up to 58% using our ReduceM algorithm.

Finally, we also demonstrate that our representation yields a practical method to use data parallelism during ray intersection to speed up tracing of single ray. By intersecting one ray against multiple edges of the triangles in parallel, we can increase the overall rendering speed by up to 90% on complex models. This is important for applications that do not work well with the standard ray packet approach for data parallelism, such as path tracing.

1.6.2 Interactive ray tracing of deformable models using bounding volume hierarchies

We also present an approach for ray tracing of deformable models. We use refitting approaches for object hierarchies such as BVHs that can maintain the hierarchy for deforming geometry and introduce a novel approach to automatically detect hierarchy quality degradation due to refitting. In practice, this approach fully detects degradation even for worst case models with highly changing connectivity where plain refitting results in a performance decrease to a fraction of the original rendering speed. In addition, the hierarchy quality control is tightly integrated into the refitting process and adds virtually no overhead to the original update algorithm. We also show algorithms for fast and data parallel ray packet intersection of BVHs in order to achieve interactive performance. Our method achieves a speedup of $2 - 2.5\times$ using data parallel operations on current CPUs and in practice can achieve interactive frame rates (larger than 10fps) on all our dynamic benchmark scenes with hundreds of thousands of triangles on a current multi-core CPU system.

We also present analysis and comparison of BVHs against other acceleration structures, specifically kd-trees and other object hierarchies such as s-kd-trees. Our results show that BVHs can provide similar performance to kd-trees when using ray packet

approaches, without some of the disadvantages that these packets have in kd-trees. In addition, BVHs tend to be more compact and predictable in memory footprint. Compared to more light-weight object hierarchies, s-kd-trees provide better performance due to less expensive intersection. However, we demonstrate that refitting approaches do not work well on these object hierarchies and quality degradation in performance occurs much quicker.

1.6.3 Parallel algorithms for hierarchy operations on GPUs

Another aspect of our work consists of algorithms for operating on bounding volume hierarchies on massively parallel architectures such as GPUs. We present the first algorithm for building object hierarchies optimized for ray tracing on current GPUs by using both data and thread parallelism. Then, we show how to improve its scalability on complex models through a new approach that eliminates the bottleneck in the early stages of the algorithm by substituting with a linear-time build. Our results show that the approach scales well even for large models and provides comparable performance to multi-core approaches [148] for hierarchy construction. We also present an analysis of performance scalability in terms of future GPU architectures and determine that the algorithm is computation bound (as opposed to bound by memory accesses) and thus is likely to scale well. Overall, the construction is fast enough to allow interactive rendering of complex scenes with hundreds of thousands of triangles while rebuilding the hierarchy at each frame.

In addition, we show a parallel approach for rapidly refitting the hierarchy for deformable geometric objects on GPU-like architectures. Due to high memory bandwidth, this approach performs several times faster than on a CPU and is about an order of magnitude faster than our fast hierarchy construction. This means that even for complex models the hierarchy can be maintained in milliseconds on current GPUs.

1.6.4 Interactive sound simulation in dynamic scenes

We present an algorithm to extend the fast BVH ray packet and refitting approaches to perform interactive geometric sound propagation with specular reflections in dynamic scenes. In particular, we introduce a novel volumetric representation based on four-sided ray-frusta as a discrete approximation to sound beams and then show how to efficiently trace these frusta using a BVH acceleration structure. This approach has several advantages compared to traditional stochastic ray-based techniques used in sound propagation that suffer from sampling problems, and is substantially faster than beam-tracing methods that needed significant preprocessing and storage. In particular, our approach works well on more general and much more complex scenes than any previous method. In practice, our propagation algorithm can perform 3-4 specular reflections at almost interactive rates on models with up to hundreds of thousands of triangles, thus allowing moving sound sources, listeners and geometry on current multi-core CPUs.

1.6.5 Highly-parallel collision detection on GPUs

We also present parallel object-object intersection using bounding volume hierarchies on GPUs and use them for fast collision detection. We introduce a novel lightweight work balancing approach as an alternative to traditional work queue approaches that do not perform well due to the synchronization overhead of the GPU cores. Additionally, we present techniques for exploiting inter-frame coherence between collision detection in subsequent frames to increase the parallelism in intersection computation. One result of our algorithm is that on highly-parallel processors such as GPUs more tight-fitting bounding volumes such as oriented bounding boxes (OBBs) have much lower overhead in construction and refitting due to higher computational intensity, and yield improved performance as compared to simple bounding volumes such as axis-aligned bounding boxes or spheres. Overall, our results show that due to fast hierarchy maintenance and work balancing it is possible to perform self-collision computations including continuous

collision on models with tens or hundreds of thousands of triangles at performance competitive with highly-parallel CPU systems. We observe up to an order of magnitude performance improvement over prior GPU-based algorithms for collision detection.

1.6.6 Hybrid GPU shadows on massive models

Finally, we also present efficient techniques for rendering accurate shadows in massive models on GPUs which cannot be handled well by rasterization methods. In particular, we introduce an approach that combines fast rasterization-based shadows with ray tracing based visibility to correct localized errors left by the original algorithm. One contribution is the adaptation of our previous compact memory representation as a format that can be used as a data structure both for a ray tracer as well as a rasterization-based rendering algorithm to avoid data duplication. We demonstrate how to use our approach both for rendering of hard and soft shadows. In practice, our results demonstrate that it is possible to generate accurate alias-free hard shadows on complex models in about $35\% - 50\%$ of pure shadow mapping algorithms, which is about $3 - 5$ times faster than only using ray tracing.

1.7 Thesis organization

The rest of this dissertation is organized as follows:

Chapter 2: A detailed overview of previous and concurrent work related to the areas of this thesis. In particular, we review related work in interactive ray tracing, ray tracing of massive models and construction of ray tracing hierarchies.

Chapter 3: We describe our compact representation for interactive ray tracing of triangular meshes. The main focus of this chapter is on massive static data-sets. We

describe how to construct our ReduceM representation and use it for ray tracing queries including hierarchy traversal and intersection.

Chapter 4: This chapter and the next discuss issues related to ray tracing dynamic scenes. Chapter 4 concentrates on ray tracing of deformable models and shows techniques for fast ray tracing using bounding volume hierarchies that can be refitted instead of fully rebuilt each frame. We discuss issues regarding degradation of rendering performance when refitting and present our solution to maintaining BVH quality over time.

Chapter 5: The second chapter dealing with dynamic scenes. Here, we introduce our approach to fast construction of bounding volume hierarchies for more general dynamic scenes that hierarchy refitting does not support. In particular, we discuss issues and algorithms in terms of using parallelism in construction algorithms by using massively parallel GPU architectures.

Chapter 6: This chapter deals with several applications that use the previously described algorithms and representations. Section 6.1 describes a geometric simulation algorithm for sound generation that performs interactive propagation of sound waves in complex, dynamic geometric environments using the dynamic BVH system. Section 6.2 presents an approach using parallel hierarchy construction and refitting in addition to parallel hierarchy operations to perform fast hierarchical collision and self-collision of complex deforming models on the GPU. Finally, section 6.3 explores the combined use of rasterization and ray tracing algorithms on GPUs as a hybrid system for interactive generation of high-quality shadows.

Chapter 7: We conclude with restating the main results of the thesis and highlight many areas for future work.

Chapter 2

Previous work

This chapter presents a survey of related work in the area of ray tracing. First, we review basic previous work in interactive ray tracing mostly concentrating on static, complex models before moving on to related work on ray tracing dynamic and deformable scenes. In addition, we examine algorithms for fast construction and refitting of ray tracing hierarchies.

2.1 Interactive Ray Tracing

Even though ray tracing for computer graphics was introduced very early [6], it has been traditionally limited to offline rendering and simulation for a long time. However, the steady increase of computational power coupled with algorithmic advances in ray tracing hierarchy construction and fast ray intersection has recently given rise to the field of interactive ray tracing with the goal of allowing real-time implementations of algorithms such as Whitted [158] or distribution ray tracing [22]. One development has been the introduction of ray packet techniques that bundle groups of rays together during intersection with the acceleration structure and geometric primitives. If any ray can hit a node in the hierarchy or a geometric primitive, then all rays are intersected against the same node or primitive. Wald *et al.* [151] first demonstrated small ray packets on kd-trees that were also accelerated using vector units on mainstream CPUs

and resulted in several times the performance of single ray tracing. Additionally, they proved that ray packets larger than actual vector sizes (e.g. 64 rays) yielded additional improvements in performance.

The performance of ray packet approaches is dependent on ray *coherence*, i.e. the probability of the rays to hit the same or close-by objects. For example, eye rays from adjacent pixels usually have high coherence since they move in very similar directions. On the other hand, rays that are stochastically sampled on a hemisphere such as used in path tracing tend to go into very different directions, and hence coherence is low. In ray packet tracing, lack of coherence means that individual rays in a group may need to be intersected against a very different set of nodes and primitives. Since all rays are intersected with the super-set of the nodes and objects hit by any ray, work overhead is introduced in the form of additional intersection operations. If coherence is high, however, then ray packet approaches save a significant number of intersection steps as just one ray may need to actually perform the intersection. It should also be noted that the primitives to be intersected against only needs to be loaded once, thus reducing memory bandwidth.

The performance advantage of coherent ray packets was further improved upon in multi-level ray tracing (MLRT) [124]. The main feature of that approach is to use very large packets that could be subdivided into smaller groups when needed. In addition, frustum culling techniques can test the whole group for intersections and are independent from the number of actual rays in the packet. The main component of the algorithm is the entry point search that for a group of rays finds the node deepest in the tree that all the rays will intersect. When this is a leaf, then actual intersection with ray primitives can occur. Otherwise, the group is split into smaller packets to find deeper entry points in the tree. The main advantage of the MLRT method is that it exploits ray coherence to a much higher degree because it can form much larger initial groups of rays and then subdivide them as needed. Thus, for coherent work loads such as eye rays and shadows

Reshetov *et al.* demonstrated speedups of an order of magnitude compared to standard packet approaches.

Rays queries can almost always be evaluated fully independently. Therefore, parallel implementations for ray tracing are trivial and multi-threaded implementations perform well without much effort on multi-core architectures. Graphics processors (GPUs) are an example of massively parallel architectures, and due to their theoretical computational power have been a logical target for ray tracing. On the other hand, lack of programmability and architectural limitations have made the design of ray tracing methods on GPUs challenging. One early approach [17] used the GPU only to perform ray-object intersections generated by a CPU hierarchy algorithm. A concurrent method [119] used a three-dimensional grid as acceleration structure and thus was able to perform the complete ray tracing algorithm on the GPU. However, the inflexible programming interface and instruction limitations resulted in overall performance far off from comparable CPU implementations and achieved only a fraction of theoretical computational power.

Subsequent approaches concentrated on using hierarchical approaches instead as those had been shown to be highly efficient in CPU ray tracing approaches [61]. The main problem for the traversal of hierarchies on GPUs was that ray traversal algorithms essentially perform recursive queries on the tree structure and as such at some level need a stack to represent the current state of traversal. This stack can potentially be large – up to the maximum depth of the tree. Since space to store this stack on GPU cores was not available, initial implementations of hierarchy traversal concentrated on finding methods that did not rely on any state. Thrane and Simonsen [142] used a BVH with a fixed traversal order with skip pointers [135] where the information on which node to go to next is always explicitly available and no stack needed. Since this limitation prevents any front-to-back ordering, this approach only works for simple models with low depth complexity. A more advanced version was presented for kd-trees in [117]. Here, each node has precomputed *ropes*: pointers for each of the 6 sides of its bounding

box. If a ray exits a node, the respective rope then provides information on where in the tree to continue traversal without a stack. The main disadvantage of this approach is that the memory footprint of the hierarchy increases significantly because an extra 6 pointers are stored per node, with additional precomputation. The kd-restart and kd-backtrack methods [40] also work on kd-trees, but require less modifications. The kd-restart method does not require changes to the kd-tree structure and always continues from the root node whenever recursion ends, but stores a distance that allows to avoid the sub-trees already visited. Thus, it is less work efficient as it may visit some nodes in the tree multiple times. The kd-backtrack algorithm relies on parent pointer in the tree to find the next node and provides slightly higher performance at the cost of higher memory footprint.

As GPUs added more flexibility in memory and programming instructions, work concentrated on algorithms that could work with limited stacks. The kd-restart restart algorithm has been combined with a small fixed-size stack [69] to avoid most restarts and reduce the work overhead. This yielded much improved performance that for the first time was comparable to CPU systems. For BVHs, Popov *et al.* [117] showed that packet ray tracing techniques could also reduce the state for stacks since only once stack is needed for a whole group of rays. Alternatively, a modified restart algorithm using restart trails can allow stackless traversals on BVHs as well [91]. On current GPU architectures, limitations have been relaxed such that stack-based recursive traversal is usually not a problem, although some architectural concerns like scheduling still have to be taken into account [2, 113].

2.2 Ray tracing of large models

Using ray tracing on massive data sets provides several challenges mainly due to the high memory requirements. One approach to the problem has been to implement ray

tracing on supercomputers [114], large shared memory systems [138, 29] or clusters with distributed memory [27]. This allows to store the geometric primitives and hierarchy in main memory such that standard ray tracing methods can be used unaltered. For machines with limited main memory, several methods have been proposed to avoid the problem of out-of-core rendering. We can broadly classify these approaches as reordering techniques, levels-of-detail approaches and compact representations.

Reordering: Ray reordering methods change the order in which rays are traced to generate more locality in the memory accesses, thus needing a smaller working set and potentially only parts of the entire model in the main memory at any time. Reordering is possible on several scales. On a high level, just the order in which rays are evaluated is changed, but the actual ray traversal and intersection order is unchanged. The approach in [115] performs this by ordering all rays based on their origin in a 3-D grid first, then tests the rays against geometric primitives in each cell such that only local geometric data can be accessed. In a hybrid system using CPUs and GPUs [15], simple reordering is also used to schedule rays for execution on processors with only parts of the scene in local memory. Another recent approach instead intersects the rays against a very simplified version of the scene and then groups them according to approximate hit point [107] instead of origin. All these approaches can drastically reduce the working set size during ray tracing and result in a large speedup compared to standard ray tracing. However, all of them also have in common that the reordering process adds significant rendering overhead. In practice, they therefore require large ray workloads and offline processing to amortize the overhead of ray organization.

At a lower level, it is also possible to change the order of ray evaluation during hierarchy traversal and intersection tests. In breadth-first ray tracing [108] all the rays are traced through the hierarchy at the same time and intersected with the same nodes, then queued up for the next set of intersections. This means only one geometric object

is ever needed to be stored in memory, but the overhead for queuing and organizing the rays is high. In this context, all ray packet approaches can be considered a sub-class of reordering approaches on a very small scale, but the groups are too small to have a large impact. Several algorithms suitable for real-time rendering have been proposed [36, 154, 47] and work by limiting the scope of reordering to a subset of all rays. At these scales, reordering through stream compaction or other methods becomes more practical during traversal, but currently the overhead still outweighs the benefits.

In general, the space of approaches between reordering the rays and reordering the traversal steps is much less explored. There is some initial work into using ray hierarchies [125] that is motivated mostly by trying to avoid the need for building a scene hierarchy. In this approach, a simple hierarchy of the rays is used and then all geometric objects are streamed over that hierarchy to perform intersection computations. However, performance is low except for very coherent ray sets.

Levels-of-detail: Algorithms using simplified versions of object data in order to reduce the rendering workload are common in GPU rasterization systems [99]. In ray tracing, levels-of-detail (LODs) are not as easy to use since every ray may need to access a different version of the model whereas for rasterization only once choice for each object is required per frame. Nevertheless, several approaches using LODs have been proposed. For subdivision surfaces such as used in animation and modeling, multiple simplification levels are easily computed, but may take too much memory. Christensen *et al.* [20] propose a multi-resolution method for offline renderers that decides on the LOD based on ray differentials [72] and caches the generated meshes. In contrast, the Razor system [31] concentrates on the use of LODs for rendering performance and quality and not memory use. It uses lazy hierarchy construction and refinement of meshes during ray tracing and addresses issues of matching simplification levels when using LODs for secondary effects. An implementation was still significantly slower than comparable

triangle-based ray tracers. For triangle meshes, Yoon *et al.* [164] introduce a method that integrated a planar LOD representation with the kd-tree structure. Rays could choose to use a simplified version of the geometric object from any point depending on their size compared to node size, as well as a user-defined quality parameter specifying acceptable error in terms of size relative to ray footprint. This resulted in far smaller working set size and thus allowed to maintain logarithmic performance despite rendering out-of-core. Finally, the approach in [153] hides memory latency for objects not in main memory through explicit memory management where ray tracing calls to parts of the hierarchy or geometry not currently in cache would be intercepted. In that case, the result is substituted by a simple precomputed placeholder object for those parts of the scene and the real data loaded in the background without blocking. This ensures that interactive rendering speed is maintained and unaffected by disk cache misses, but requires relatively low-level interaction with the operating system. Overall, all these techniques may introduce visual artifacts and may also increase preprocessing time and overall memory footprint.

Compact representations: Finally, several algorithms have been proposed to use more compact representations for ray tracing acceleration structures. By reducing the overall memory footprint of the hierarchy, larger models can be ray traced in core. Reduced precision BVHs [101] quantize the coordinates of each bounding box hierarchically such that each only has between 4 and 16 bits (instead of 32 for single-precision floating point). Overall, this can drastically reduce the memory needed per BVH node. On the other hand, since the quantized boxes can be larger than the original ones, this approach could result in a higher number of ray-box intersections. During traversal the algorithm must first decode the bounding box coordinates back to floating point, leading to additional run-time overhead. Even though some of that overhead can be amortized using ray packet techniques [101], compressed representations still incur rendering over-

head. Another technique [21] also uses higher branching factors to reduce the number of nodes and a balanced tree layout such that no child pointers are needed, but has similar drawbacks. In addition, for models with uneven geometric distribution balanced trees commonly perform much worse than using optimized hierarchies. Neither of the approaches reduces the overhead of storing geometry data.

2.3 Ray tracing dynamic models

For any non-trivial scene, ray tracing algorithms depend on an acceleration structure to quickly determine a set of geometric objects that a given ray can potentially intersect. The disadvantage is that these structures typically become invalid as soon as objects move or deform and when new objects are inserted or existing ones removed. While there are approaches that can compute acceleration structures that are valid for an animation, they need a priori knowledge of the whole animation (i.e. all the frames and object positions) to work and typically also decrease rendering performance due to larger bounding boxes. In space-time ray tracing [49], bounding boxes are built over the whole space of the animation for each model. The approach described in [57] is more practical for complex objects and first decomposes the object into smaller meshes, then computes the animation bounding boxes for each of the meshes instead.

There are two different approaches for ray tracing of animated scenes: the first is to fully rebuild the acceleration structure for each frame and the second is to refit or update the structure in order to restore correctness for geometric objects that have moved since the last frame. The two following sections address each of these solutions. In this context, we mainly limit our discussion to hierarchical acceleration structures, but a more detailed comparison of all approaches related to ray tracing of dynamic models is available in [156].

2.3.1 Hierarchy construction for ray tracing

In ray tracing, the fundamental problem of construction is to build a hierarchical data structure from a given unstructured set of objects such as triangles. In this context there are two main issues to address: how to construct hierarchies that provide high ray tracing performance, and how to construct these hierarchies quickly. Almost all algorithms for building hierarchies perform a recursive top-down subdivision of the set of objects until some termination criterion is reached, such as a sufficiently small number of objects remaining in each group. The main differences in algorithms stem from the decision on how to subdivide the object set at each step. Early hierarchy construction implementations [126] used simple methods such as subdividing along the longest axis of the objects' bounding box, or splitting to get an equal number of object on each side to result in a balanced hierarchy. However, further analysis showed that this yielded sub-par ray tracing performance. The surface area heuristic (SAH) [50, 100, 61] is a greedy approach that minimizes a cost function at each split in order to find the one with the least expected ray intersections and thus best expected ray tracing performance. During each subdivision it minimizes a cost function that models the expected number of ray intersections for the subdivision. This minimizes the overall surface area of all tree nodes, and, since the surface area of a bounding volume is proportional to its probability of being hit by a ray from a uniformly sampled direction, reduces the number of expected ray-node intersection operations. Results have shown that especially for scenes with irregular object distribution (such as architectural and CAD models) this can result in a 2-3x higher rendering performance [61, 149]. While the SAH was initially introduced for the kd-tree data structure, it has also been applied to others such as BVHs [93, 152] and spatial kd-trees [163].

Evaluating the SAH cost function can be slow because it usually involves sorting the objects on each axis at least once as a preprocessing step to finding all the split candidates [155]. A recent development has been to use approximate SAH techniques

instead [116, 71, 150]. Here, the SAH cost function is only computed for a small set of candidates instead of all possible ones. Based on the results of those sample values, an approximate overall cost function for the split can be constructed by standard interpolation techniques, and the minimum value of that function is used to find the best split available. In addition, data parallel operations can be used to evaluate multiple candidates at the same time [71]. Results from these methods have shown that this speeds up overall construction drastically while only sacrificing a negligible fraction of overall ray tracing performance. As an alternative, other very fast object hierarchy splitting techniques were also developed [163]. Here, initial splits are performed using very fast spatial subdivision similar to an octree.

Another avenue for increasing construction speed is to parallelize the tree construction. Early work in [116] distributes splits to parallel threads on a multi-core CPU system, but the authors found that the bottleneck of the algorithm was the lack of parallelism for the first splits at the top of the tree. Since these early splits also are the largest in terms of number of objects to process, the least parallelism is available where the most work is done. For that reason, their implementation would only achieve sublinear speedups even with a high number of cores. This problem was addressed in later work [131] for kd-trees with the introduction of a initial clustering step that coarsely sorts objects into larger groups. This clustering then provides the input to the real subdivision build. Since the splits of each group can be handled independently, this provides enough parallel splits to achieve enough available parallelism for better scalability. In addition, the clustering process itself is also a parallel operation. Subsequently, this approach provides linear scalability with a small number of cores, but does incur a small hierarchy quality penalty for using a simplified clustering method for the top levels of the tree. This penalty is also dependent on the number of parallel cores used. A similar approach is presented in [150] for BVH construction. The authors investigate a initial clustering approach as in [131] and a coarse grid sort as a faster alternative, and see

similar scalability results. However, for both approaches scalability falls beyond 4 cores which is most likely to be attributed to limitations of memory bandwidth, which remain an issue in hierarchy construction. Choi *et al.* present an improved multi-core kd-tree construction algorithm [19] that avoids some of the quality trade-offs made in previous parallel algorithms, but also achieve only about 7x scaling on a 16-core architecture. On GPUs, Zhou *et al.* [166] have demonstrated an algorithm for building kd-trees in parallel. Using work queues and a breadth-first construction, splits are assigned to GPU cores and then the individual split operations performed using vector units. To avoid the problem of insufficient parallelism for the first splits, they process the top levels of the tree with a very simple split and only use SAH splitting at the lower levels. Their results show that parallel construction of hierarchies on programmable GPUs can achieve comparable performance to multi-core CPU systems. For GPU BVH construction, Pantaleoni and Luebke [112] improve the LBVH algorithm [94] in performance and hierarchy quality by introducing several ideas: first, they introduce an improved version of the LBVH sorting algorithm that uses hierarchical information to achieve a 2-3x speedup over LBVH. Second, they also use LBVH information for a hybrid SAH builder. However, due to the nature of the LBVH construction the approach still produces the best results only on models with relatively uniformly distributed geometry such as scanned models.

Finally, another solution is to eliminate the need to fully build the acceleration structure by using lazy construction where the data structure is only built as it is accessed during the ray intersection. Waechter *et al.* [163] investigate this for complex models and find that it may significantly construction times. Similarly, the Razor system [31] builds all acceleration structures lazily including generation of subdivision surfaces. Overall, lazy construction has the advantage of reducing the time-to-image for rendering and may also reduce the overall storage requirements of the hierarchy. However, it may be less efficient if most parts of the scene are intersected by rays, such as with indirect

illumination and other less coherent effects. Combining parallelism and lazy construction also still poses problems that may be harder to solve.

2.3.2 Hierarchy refitting approaches

As an alternative to rebuilding, for object hierarchies such as BVHs it is also possible to modify the existing hierarchy to account for moving objects. While it is possible to insert and remove objects into object hierarchies [34], doing so may not be much faster than a full rebuild if many objects change per frame. In general, most methods are limited to deformable objects where only positions but not the number of objects changes. Overall, the refitting operation on a hierarchy consists of recomputing all bounding volumes in the tree to fit to the deformed geometric primitives, but leaving the actual tree structure unchanged. In practice, this can be done through a post-order traversal of the tree where at the leafs bounding volumes are fit to the geometric objects and new volumes are then propagated upwards. For ray tracing, this has shown to work well as long as object deformation is coherent and connectivity constant [152]. If this is not the case, the ray tracing performance of the updated hierarchy may decrease rapidly. Some methods have been proposed to handle this problem. If the animation with object positions is known in advance, then it is possible to minimize the impact by computing the BVH for each frame and then using the one that fits all frames best as input [152]. Selective restructuring [165] is able to partially rebuild parts of the tree as determined by a cost/benefit analysis, but the additional overhead leads to only small overall improvement in performance compared to refitting. The approach in [46] tries to identify different types of motion in the lower levels of the tree and uses that as a criterion for refitting or partial rebuilding. Overall, hierarchy refitting has been shown to be about 1-2 orders of magnitude faster than a rebuild [152], but does not scale linearly to multiple CPU cores for larger scenes since it can be memory limited.

Chapter 3

Massive models

One of the reasons that rasterization algorithms provide high performance for rendering massive models has been the almost linear memory access pattern and small working set size. However, as was explained in chapter 2, such a pattern cannot be directly replicated for ray tracing. On the other hand, a good alternative in order to accelerate ray tracing is to make the geometric object representation and acceleration structure sufficiently compact by lowering the storage overhead and therefore being more memory efficient. In this chapter, we describe a compact representation for interactive ray tracing of massive triangular models that significantly lowers the overall memory footprint and thus enables in-core ray tracing of much larger models. Similar to the development of many efficient rasterization approaches, we exploit the connectivity between the triangles to reduce storage overhead and even use that connectivity to partly represent the hierarchy.

3.1 ReduceM Representation

3.1.1 Overview

Our representation is based on the idea of a two-level hierarchy (see Fig. 3.1). Overall, the model is decomposed into smaller meshes that are each stored in our representation. Each of these meshes includes both the hierarchy and geometry information necessary

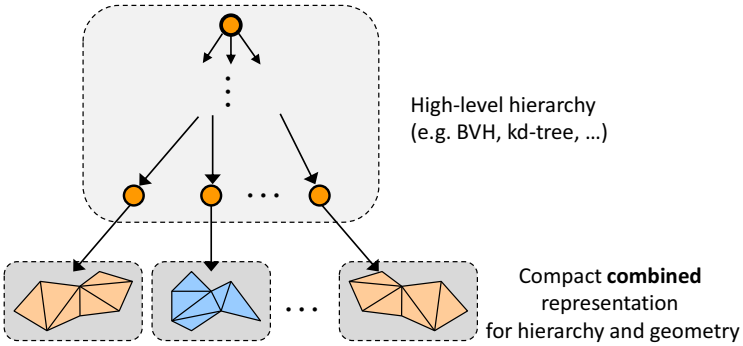


Figure 3.1: **Using a two-level hierarchy:** *Our representation is based on a compact representation for meshes and hierarchy at the bottom levels and a top-level hierarchy that is built on only the meshes and thus much smaller than a hierarchy built on the original triangles.*

for ray traversal and intersection, but in a very compact format that is still efficient, as described below. The top-level hierarchy is then built on top of these meshes as opposed on the actual underlying triangles. This results in a much smaller top-level hierarchy since there are far less meshes than original triangles. In general, the top-level hierarchy can be any choice of acceleration structure such as kd-trees, BVHs or grids. In this further discussion, we assume that a BVH is used for simplicity.

The ReduceM representation has several components that will be described in turn. First, the actual compact representation is based on triangle strips and encodes the geometry in a format accessible to a ray tracer, but also stores an object hierarchy with almost no overhead. Second, we describe efficient methods for traversing this compressed hierarchy and intersecting with actual triangles. In particular, both ray packet approaches and single ray traversal can be performed very efficiently in this context. Finally, we discuss an algorithm for actually constructing our representation from unstructured triangles that provides good performance for ray tracing by modifying the triangle stripification algorithm.

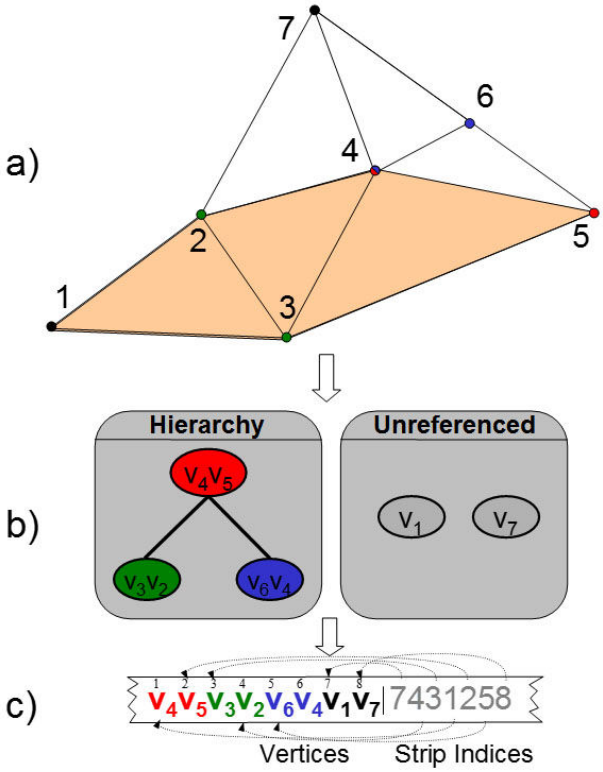


Figure 3.2: **ReduceM representation:** *This figure shows a triangle strip consisting of 7 vertices (see **a**)). Our ReduceM representation consists of two main components: 1) a list of vertices and 2) a list of vertex indices referring the vertices. We implicitly encode a balanced s-kd-tree structure (see **b**)), where each split along an axis partitions triangles into halves. By laying out the vertices referenced in the strip carefully, we can encode the bounds of the s-kd-tree (see **c**)). The actual strip is then defined by indices after the vertices. Vertices not referenced in the hierarchy (here: 1 and 7) are stored afterwards.*

3.1.2 Representation

We now present the details of the actual representation. An example is shown in Fig.3.2. The ReduceM representation has three main components to store: 1) vertices (e.g., vertex coordinates), 2) the triangle connectivity information, and 3) the hierarchy encoded on the mesh. In order to preserve maximum locality of data access during traversal and intersection, we store all the vertex coordinates directly in the strip without using a global vertex list. Encoding the vertices directly instead of using indirect access through the global vertex list can duplicate the vertices that are shared between different strips, but reduces non-local memory access. As a result, it has a comparable or even better memory usage (such as cache misses) compared to indexed vertices. This benefit is mainly obtained by eliminating one indirection during ray tracing and preserving locality. Even though vertex coordinates are stored locally in the ReduceM representation, we store the actual triangle strip via vertex indices referring to vertex coordinates in the ReduceM representation. Each vertex index is encoded in one byte, which limits the maximum strip length. However, we have found that in practice useful strips longer than 255 vertices are hard to extract in many real-world models. On the other hand, vertices can often be referenced multiple times in one strip. For example, to create a valid triangle strip ordering from a sequence of triangles, it is often necessary to introduce edge swaps in a triangle strip in order to 'flip' the edge order in the strip by introducing the same edge twice (i.e. for the edge AB by having the sequence ABA) and hence creating a zero area triangle as well as an additional vertex reference. Therefore, encoding the triangle strip with vertex indices of small size will reduce the impact of such swaps in the ReduceM representation.

We use a balanced spatial kd-tree (skd-tree, see also chapter 4) [110, 163] for the strip hierarchy, which is a compact object hierarchy where instead of storing a full bounding box per node we only store two coordinates for a given axis. One advantage of using left-balanced trees (i.e. a balanced tree where the left sub-tree is always filled first) is

that they can be stored in an array in order by level where the location each node's children can be computed directly from its offset without needing an explicit pointer. If both children are stored next to each other, then the children of a node at offset i are at $2i$ and $2i + 1$, respectively.

Since we do not need child pointers, the only data left to store per node are the bounds. Each node of the spatial kd-tree is represented by the upper and lower bounds for the side of the split, respectively. Our observation is that the bounds for each split always coincide with one or more vertices on each side of the split. By using vertex indices for encoding the triangle strip, we are free to order the actual vertex coordinates in the layout. Therefore, we reorder coordinates of vertices such that every two consecutive vertices represent each split of the s-kd-tree. Based on this formulation, we implicitly define a left-balanced s-kd-tree from the underlying ReduceM representation (see Fig. 3.2). Each node of the hierarchy also needs to store the split axis for that node, using 2 bits. While it is possible to list this information separately, we store all the vertex coordinates relative to the strip's bounding box known from the traversal, and then encode that information into the sign bits of the vertices.

In general, the upper and lower bounds of the strip hierarchy do not reference all the vertices in the strip. Therefore, we need to list all the unused vertices as well (e.g. vertex 1 and 7 in Fig. 3.2). In addition, there are cases where a vertex may need to be used twice in a strip hierarchy (such as vertex 4 in our example). Therefore, there is extra memory overhead from storing vertices twice in the representation. We minimize this overhead by searching for multiple split vertex candidates and always considering previously unused candidates first. We have found that for our test models this method effectively minimizes the overall overhead in our test models (see Table 3.1.) Note that this is the only actual overhead for storing a hierarchy on top of the geometry representation.

Model	Tris	Build time	Hierarchy overhead
Powerplant	12.7M	5m	1.58%
Double Eagle	82M	33m	1.95%
Puget Sound	134M	36m	3.24%
Boeing 777	364M	1h50m	2.71%
St.Matthew	372M	1h36m	3.56%

Table 3.1: **Construction statistics:** *The build time column shows the overall time taken for construction including stripification and high-level hierarchy for our benchmark scenes. Timings are on the system described in 3.4 and multiple parallel threads were used to speed up construction. The last column shows the overall storage overhead in terms of duplicated vertices as a fraction of total vertices in order to implicitly store the strip hierarchy.*

3.2 Ray tracing using the ReduceM representation

We now describe how to use this compact representation for fast ray tracing.

3.2.1 Hierarchy traversal

The ReduceM hierarchy is essentially a spatial kd-tree. Therefore, the triangle strip can be traversed in a very similar manner to the spatial kd-tree (see [163], also illustrated in Fig. 3.3). Starting at the root node, the bounding box of the mesh is split in the axis stored in the node. Given a ray, the distances d_L and d_R to both split planes can be tested to determine whether it hits the left, the right or both children, where the order is given by the ray's direction.

If the ray intersects both the children, the near one – as determined by the ray's direction – is traversed first and the far one is pushed on a stack. The children are processed in the same manner by updating the bounding box from the parent with the child's respective split plane. Since the hierarchy is subdivided according to fixed rules and the number of triangles in the mesh is known, the traversal just needs to keep track of how many triangles are left in the current sub-tree to know when it encounters a leaf. This is performed by updating the interval in the triangle strip representing the triangles whenever the traversal jumps to another node. The pseudo-code implementing

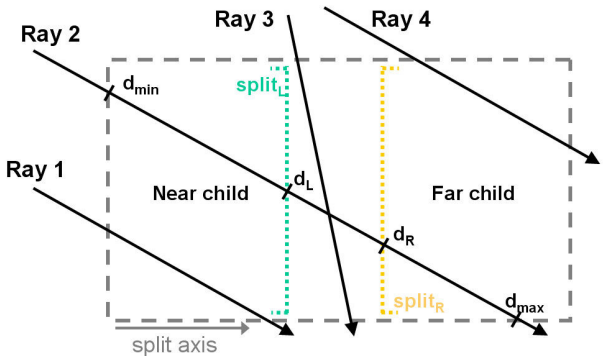


Figure 3.3: **Traversing a node:** *Given an inner node of the mesh hierarchy, there are two split planes associated with it. The ray hits the left child if the distance to the left plane d_L is larger than the node entry distance d_{min} and hits the right child if the node exit distance d_{max} is larger than d_R. For ray 1, only the near child is traversed, for ray 2 both need to be traversed and ray 4 only traverses the far child. Note the special case for ray which traverses neither of the two children.*

this traversal is shown in Algorithm 3.1.

3.2.2 Intersection Computation

At every step in the traversal, the ray tracer can decide to intersect with the triangles contained in the node, which are defined by the current interval in the triangle strip sequence. The naïve approach would be to test every triangle by itself using a standard triangle intersection algorithm, but that would ignore information available as part of the strip representation. Instead, we take advantage of the connectivity information. Specifically, we use an edge-based intersection that tests the ray for containment using Plücker coordinates[133], which allows us to test the orientation of a ray relative to an edge. Given three edges defining a triangle in a consistent order (clockwise/counter-clockwise), the ray intersects the triangle if and only if the signs of the Plcker edge tests match. Given an interval in the triangle strip, we can compute all the edge tests directly and then test each consecutive set of three edge results for intersection with the respective triangle. Since there are shared edges, this is more efficient than testing each of the triangles. In particular, we can easily test 4 edges at the same time by using

Algorithm 3.1 Traversal of the ReduceM hierarchy

```
node ← root, left ← 0, right ← #indices - 1
[d_min, d_max] = ray.intersectWithBoundingBox()
while node != NULL do
  while (right - left + 1) ¿ 2 do
    d_L ← ray.distToNear(node.axis, node.split1, node.split2)
    d_R ← ray.distToFar(node.axis, node.split1, node.split2)
    if d_L < d_min then
      if d_R > d_max then
        [node,left,right,d_min,d_max] ← stack.pop()
      else
        [node,left,right] ← node.farChild(ray,node.reverse)
      end if
      continue
    else if d_R ≤ d_max then
      stack.push(node.farChild(ray,node.reverse), max(d_R, d_min), d_max)
    end if
    d_max ← min(d_L, d_max);
    [node,left,right] ← node.nearChild(ray,node.reverse)
  end while
  intersectStrip(left, right)
  [node,left,right,d_min,d_max] ← stack.pop()
end while
```

SIMD instructions on current CPUs. This is particularly effective when tracing just one ray because we utilize data parallelism.

3.2.3 Ray Packets

Both the traversal and the intersection algorithms described above can be extended to handle ray packets. For traversal, the algorithm given above changes in that a sub-tree is traversed if the bounding box is intersected by any of the rays. For intersection, the edge tests have to be performed for each ray in the packet. However, if the rays in the packet share the same origin, the Plcker intersection can be optimized to reuse the coordinate computation, as presented in [11].

An important issue for ray packet tracing on massive and complex models is that rays can be very incoherent at the lower levels of the hierarchy due to small triangles and the geometric detail. This leads to traversal and intersection steps to have one or very few "active" rays, i.e. rays intersecting the current sub-tree. ReduceM allows us to detect those cases and switch to edge intersection whenever appropriate. Thus, when the number of active rays becomes smaller than a certain threshold, we switch to single ray intersection for all the active rays. This improves performance because of more efficient use of data parallelism.

3.3 Strip-RT: Stripification for Ray Tracing

The previous sections have so far assumed a subdivision of the mesh into suitable triangle strips. However, the input for ray tracing is usually a mesh in the form of unordered triangles. In this section we present an algorithm for stripification – i.e. decomposition into triangles – with a focus on constructing strips optimized for ray tracing. The goal is to lower the storage overhead and achieve high run-time performance during ray tracing using our representation. The ReduceM representation consists of a triangle

strip and a low-level strip hierarchy implicitly encoded in the triangle strip. The high-level hierarchy is computed based on the low-level hierarchy. Therefore, the efficiency of ReduceM basically reduces to computation of triangle strips from an input mesh. In order to compute optimized triangle strips for ray tracing, we consider the following two properties:

- **Triangle strip length:** We can achieve a higher compression ratio as the length of the triangle strip increases. As a result, we can further reduce both the storage overhead and the memory overhead of ReduceM.

- **Spatial coherence of the balanced hierarchies:** The balanced s-kd-tree implicitly encoded on top of the triangle strips should have high spatial coherence in order to reduce the number of intersection tests with nodes of the s-kd-tree during ray tracing. It is widely known that we can achieve this goal by considering the surface-area heuristic (SAH) during hierarchy construction [50, 100, 61].

Prior algorithms to compute triangle strips – such as Hoppe's rendering sequence [67] – satisfy the first property related to the length of the generated triangle strips. These techniques work well for rasterization where the main goal is to achieve high GPU vertex cache utilization during rasterization. However, these approaches do not address the issue of computing tight fitting hierarchies on top of triangle strips and, therefore, do not address the problem of achieving high spatial coherence between triangle strips for ray tracing.

3.3.1 Hierarchical Triangle Strip Computation

Most acceleration hierarchy construction methods for ray tracing use top-down methods while considering the SAH metric at each split. Since we implicitly encode our s-kd-tree from the constructed triangle strip, we design our triangle stripification method, Strip-RT, to construct triangle strips considering the SAH. Overall, the triangle strip

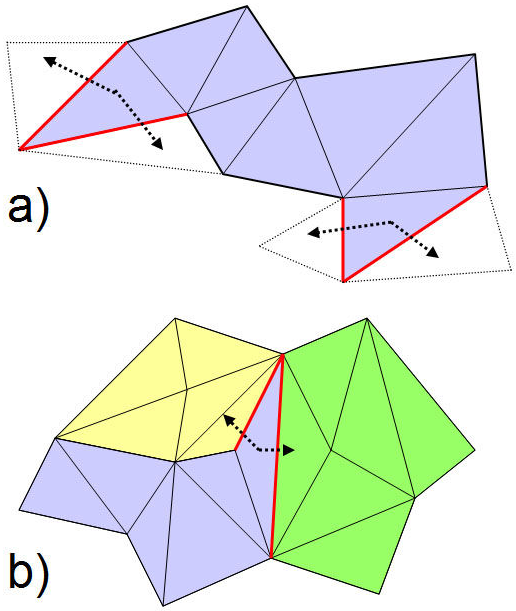


Figure 3.4: **Strip ordering:** This figure illustrates the strip ordering in the stripification algorithm. a) Each given triangle sequence can have up to four open edges (shown in red) where combination with other sequences are possible. b) The blue triangle sequence has two possible other sequences it can be combined with on its two open edges.

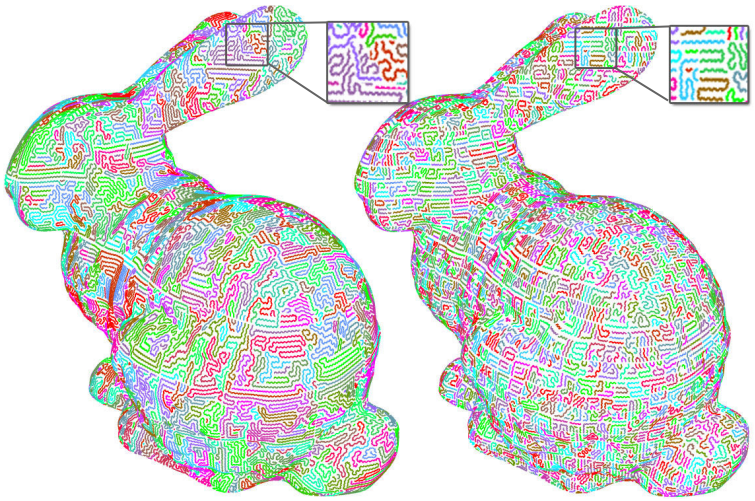


Figure 3.5: **Stripification for ray tracing:** *Triangle strips created for rasterization (such as on the left) are often relatively wide-spread and therefore have a high surface area compared to their size. Our stripification (on the right) creates strips that are both compact in terms of surface area and have sub-strips that are themselves compact. This creates higher traversal efficiency for ray tracing.*

computation algorithm is given an input mesh and produces one or more triangle strips from it using the following steps:

1. **Adjacency computation:** Usually the input mesh is not given as a graph with adjacency information but a list of unordered triangles. We find all shared vertices and edges between triangles and then record all the adjacency information in the mesh.

2. **Partitioning:** Given the mesh, the partitioning step recursively splits it into an initial object hierarchy optimized by the SAH metric.

3. **Ordering:** The ordering step takes the initial hierarchy and iteratively computes one or multiple triangle sequences optimized for ray tracing.

4. **Strip output:** Finally, using each sequence of triangles the strip output step produces a list of indices defining each strip in the ReduceM format.

Of these, 2 and 3 are the main steps in the algorithm and will thus be described in detail in the following text. Computing adjacency information in step 1 is relatively simple as long as the triangles are specified with vertex indices; otherwise, duplicate vertices have to be detected by analyzing the vertex coordinates. Computing a list of indices specifying a triangle strip from a sequence of triangles has also been discussed in depth in previous work (e.g. [38]) and we opt for a similar implementation. Note that during the partitioning and ordering steps, our algorithm may not produce just one connected triangle strip from the input mesh. In general,computing a single connected triangle strip of a mesh is equivalent to computing a Hamiltonian path, which is a NP-complete problem [30], and even if such a strip exists, it might not be a good choice for ray tracing applications.

Partitioning: We recursively partition the mesh contained in a node of the hierarchy into two sub-meshes, starting with the input mesh and ending when only one triangle is

left. During each partition, we find the best split of the triangles in the mesh by using the SAH metric and choosing the partition with the lowest cost. Thus, the partitioning step is almost the same as building an object hierarchy such as a BVH on the input mesh (e.g. as described in [145]). In addition to generating the hierarchy, we also store an inverse mapping for each triangle that records which hierarchy node it is referenced in (since this is an object hierarchy, only one node can do so.)

Ordering: The purpose of the ordering step is to find actual triangle sequences in the input mesh that can later on be converted to triangle strips. In order to find good strips for ray tracing, the previously computed hierarchy is used to provide hints on determining good pairings of sequences. Our ordering algorithm iteratively combines existing triangle sequences into longer sequences. As a first step, we consider all triangles as a triangle strip of length one and place them into a list of active sequences. For each sequence, we record the list of triangles that defines the sequence, as well as the adjacency information: since each active sequence has two potentially 'open' edges at the end (see Fig. 3.4 for illustration), 4 pointers to the adjacent triangle at those edges (or to a null object if not a shared edge) are sufficient. In addition, we also maintain a reference to a node in the hierarchy built in the previous step, initialized to leaf nodes for the initial sequences. Finally, we maintain the list of triangles in the mesh such that each triangle also has a pointer to the sequence that it is currently contained in, which allows us to find the neighboring sequences based on the triangle adjacency information.

At each iteration in the algorithm, we look at each sequence in the list of active sequences and then evaluate all possible pairings with other active sequences using the adjacency information. All sequences that have no further possible combinations are placed in the output sequence list and removed from the active list. For many sequences, however, there will be multiple possibilities for combination and we want to find the one that will most likely produce strips that are best for our purposes. To do so, we rank

all possible combinations with other strips according to two factors: first, since each sequence is associated with a node in the initial hierarchy, we test the path distance in the tree between the two nodes and prefer the combination that has the shortest one. Intuitively, this rewards combining strips that would have been split similarly in the optimized build and thus presumably are a good choice. However, we also use a second criterion by considering the *harmonic mean* between the number of triangles in each of the two sequences. The reason for this is that we later use a balanced hierarchy on the ReduceM representation and thus do not have a choice in where to put the split inside the strip. By choosing combinations of roughly equal relative size, it is very likely that the implicit hierarchy will have a split that corresponds to the two strips we are combining. For example, assume that we have two sequences with 130 and 10 triangles and another pair with 70 and 70 pairs. Although the arithmetic mean (i.e. 70) is the same, the pair with 130 and 10 connectivity pairs has a harmonic mean of 18.5 whereas the other is 70. Overall, when determining the best combination we weigh both the relative tree distance as well as the harmonic mean (relative to the total number of triangles in the sequences) equally and then pick the combination with the best results.

To merge triangle sequences, we concatenate the triangle index list of both sequences and then can easily find the adjacency information for the new strip. However, we also need to find a tree node associated with the new strip. To do so, we find the nodes associated with the two individual sequences and then assign the lowest common ancestor to the new strip. The algorithm terminates when at the end of an iteration there are no more active sequences left, i.e. no more possible combinations of sequences available. However, we have also found that it can be useful to introduce a user-specified maximum strip length parameter (please also see the discussion in section 6) that prevents sequences from growing above a certain size. The reason for doing so is that the algorithm may otherwise produce some very long strips while at the same time keeping others very short. We have found that it is more desirable to keep sequence lengths

Model	Tris	Memory (total)			Performance (fps)		
		STRIPE	ReduceM	Tipsification	STRIPE	ReduceM	Tipsification
Powerplant	12.7M	350 MB	272 MB	271 MB	28.57	31.25	29.05
Double Eagle	82M	2672 MB	1818 MB	1795 MB	11.90	12.5	11.31
Puget Sound	134M	2736 MB	2834 MB	2687 MB	7.58	12.05	6.45
Boeing 777	364M	12128 MB	8914 MB	8894 MB	3.75	4.76	3.8
St.Matthew	372M	7531 MB	7290 MB	7002 MB	3.36	4.81	3.15

Table 3.2: **Comparison:** *We single out the effect of stripification on performance and memory for the same ReduceM representation. Performance numbers are for primary visibility at* 1024^2*. Stripification is performed with STRIPE [38] and with our Strip-RT algorithm.*

more uniform for performance.

3.3.2 Massive model stripification

In order to apply our algorithm to massive models and find smaller input meshes for the stripification algorithm, we first decompose an input model into small chunks, each of which fits into main memory. Then, we apply our algorithm to the chunks, each of which can also be processed in parallel on multiple processors to speed up the stripification process, with a merge operation at the end. Table 3.1 shows information on typical construction times for our benchmark models, using parallel processing on a 16-core system.

3.4 Results

We implemented the ReduceM algorithm in a full ray tracing system running on a Intel Xeon machine with 16 cores (X7350 at 2.93 GHz) and 8 GB of RAM running on 64-bit Microsoft Windows. We have used different models with various characteristics for our benchmarks. These data sets are selected to provide examples from different application areas including scientific visualization, terrain visualization, CAD and architectural environments, and scanned models. Parameters were set such that triangle strip length was limited to at most 60 triangles and strip hierarchy traversal was set to terminate

Model	Tris	Vert.	ReduceM		kd-tree			BVH		
			Total	Vertices	Hierarchy	Total	Tree	Geometry	Total	Tree
Powerplant	12.7M	11M	272	126	171	983	496	486	875	389
Double Eagle	82M	77M	1818	884	1047	7118	3373	3147	6241	2496
Puget Sound	134M	67M	2834	768	1961	11164	6044	5120	9216	4095
Boeing 777	364M	208M	8914	2389	6525	n/a	n/a	14219	25059	11137
St.Matthew	372M	186M	7290	2138	4985	n/a	n/a	13921	25595	11375

Table 3.3: **Memory footprint:** *We show the effect on memory consumption of using ReduceM compared to kd-trees and BVHs for several complex models (all numbers in MB). The memory used by ReduceM can be split up in the uncompressed vertex data as well as the high-level and strip hierarchy representation including the connectivity. The BVH and kd-tree have the same storage for the triangle geometry, but different footprint for the hierarchy.*

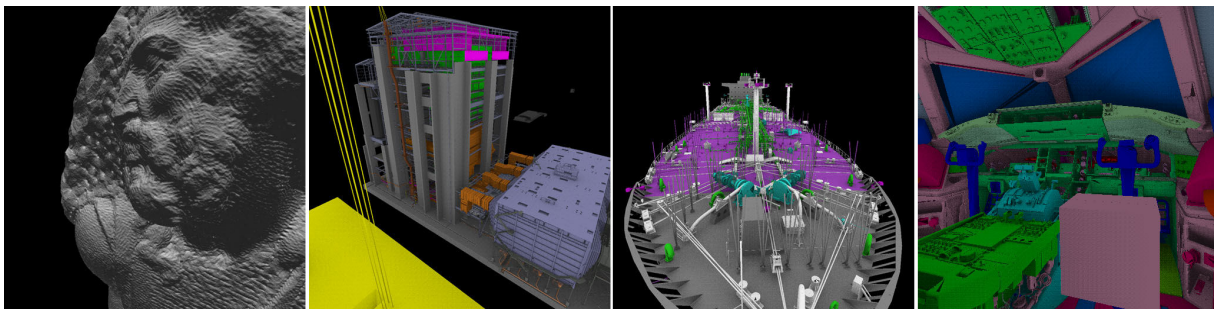


Figure 3.6: **Benchmarks:** *Screenshots from massive models used in the paper. From left to right: St.Matthew (372M tris), Power plant (12.7M tris), Double Eagle tanker (82M tris), Boeing 777 (264M). Our ReduceM representation decreases the memory needed for rendering these scenes by a factor of 3 to 5 over standard representations.*

when reaching 2 triangles.

We measure the improvements both in terms of overall memory footprint as well as the reduction in hierarchy and connectivity size alone. Table 3.3 summarizes the results for our benchmarks. We also compare the memory and rendering speed to a pure kd-tree [149] and AABB bounding volume hierarchy implementation [93, 145], both of which are popular solutions in interactive ray tracing. The kd-tree implementation uses a standard 8 byte/node representation [153], while each BVH node takes 28 bytes. All the hierarchies were built using the surface area heuristic with automatic termination criterion. Also, performance results were obtained using ray packet traversal and intersection. Note that performance results for these comparisons are not available for some of the larger models since the total size exceeded main memory size and performance results would therefore have been very low. The triangle geometry was stored in a standard intersection format [153] using 40 bytes per triangle.

In order to verify the benefits of our strip construction method, we measure memory requirement and run-time performance of our ReduceM method when using triangle strips computed from Strip-RT and a standard rasterization-oriented stripification algorithm STRIPE [38] (see Table 3.2). Strip-RT achieves up to 58% run-time performance improvement over STRIPE.

There are newer methods for stripification rasterization that are geared towards real-time construction, but as a side-effect also offer more spatial locality than previous approaches such as STRIPE. In particular, the approach in [127] starts out with triangle fans around vertices and computes a path around the fan, then combines multiple triangle fans together to form a longer triangle strip. By construction, triangles in a fan are relatively local, which satisfies some of the criteria described above. Interestingly, the authors also subdivide the model ahead of stripification by clustering the triangles, which is similar to the hierarchy construction step we use in the Strip-RT approach.

To evaluate the differences, we have implemented the local "tipsification" approach

described in [127], but instead take the same input meshes that we use as input to our stripification algorithm. We then run their algorithm to construct the actual triangle sequences, followed by the construction of the ReduceM representation. Our results are in Table 3.2. In general, we found that the tipsification approach can construct very long triangle sequences if allowed to – and typically longer ones than Strip-RT, since the criteria for "joining" strips are much more lax. On the other hand, the algorithm often chooses sequences of triangles that only share a common vertex, but not a common edge, whereas Strip-RT by construction will only choose edge-adjacent triangles. While this is not a problem for the intersection and traversal algorithm since we allow null edges by repeating vertices, it can reduce performance due to less edge-sharing and useless computation for empty edges. The results reflect this: overall, tipsification can actually reduce memory use slightly because we can find longer strips, but performance is actually comparable or lower than STRIPE (except for 777 where the low connectivity equalizes the playing field.) It would be interesting to investigate whether the algorithm can be changed to construct sequences that have better edge adjacency.

3.5 Analysis and Comparison

Any mesh compression method for ray tracing can reduce three different parts: vertex data, connectivity and hierarchy. The results from the ReduceM algorithm clearly show that we are successful in reducing the memory footprint of both connectivity and hierarchy, but even with the optimized stripification we still incur some rendering overhead compared to a fully optimized standard hierarchy. In particular in architectural and CAD scenes with strongly varying triangle sizes there still is a higher performance difference, while other models are very close. However, in the common situation where memory is in fact limited, being able to represent all ray tracing data in main memory is invaluable.

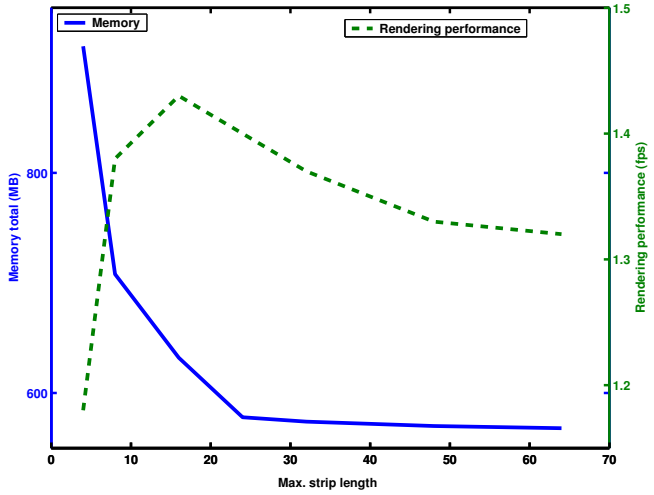


Figure 3.7: **Effect of strip length parameter:** Results for ReduceM on the Lucy model when modifying the maximum strip length parameter in construction. Both performance as well as memory footprint improve with increasing strip length. However, at higher limits memory savings decrease as only some longer strips are found and performance decreases slightly.

As discussed in section 4.1, it can be useful to limit the maximum triangle size during construction to avoid unevenly distributed strip sizes. In general, we find that due to the balanced split limitation even our optimized stripification algorithm cannot guarantee the same hierarchy quality as for example a common SAH hierarchy, and thus performance decreases slightly at some point as triangle strip length restrictions are lowered. Figure 3.7 visualizes this at the example of the Lucy model where the ReduceM representation was generated using different length limits ranging from 4 to 64 — note that this does not mean that all the strips are that size; in fact, the average strip length is usually much lower.

Comparison: Approaches to compress the hierarchy used for ray tracing were introduced in [101, 21]. By reducing the number of bits used to encode the bounding boxes, the memory complexity can be reduced drastically. However, there are two main implications. First, decoding of the coordinates has to occur at run-time and adds some overhead to traversal, although Mahovsky [101] shows that the effect can be limited by using ray coherence approaches. Second, quantizing bounding boxes inherently slows

down ray tracing since it enlarges the surface area of the boxes and thus increases the average number of traversal steps per ray. Reshetov has recently introduced a vertex culling method [122] that can reduce memory requirement of hierarchies. The vertex culling algorithm can efficiently intersect with relatively large kd-tree leaf nodes (i.e. with a much higher number of triangles than the usual 2-4 triangles per a node) and thus the overall hierarchy is smaller since fewer nodes are required. Geometry is still stored as usual. Since the paper was not focused on massive models, it is hard to directly compare both approaches due to lack of data. However, it is possible that vertex culling could be combined with the ReduceM approach as an alternative strip intersection method.

Limitations: Our method has certain limitations. First, ReduceM reduces memory requirements only for connectivity and hierarchy information, but not vertices. This puts a hard limit on the achievable overall compression rate since for massive models vertex data can make up a large part of the overall storage. Compression is also strongly dependent on available connectivity, i.e. if the input model is a fully disconnected set of triangles, no improvement is possible. For example, we have found connectivity to be a problem on the Boeing 777 model: vertex coordinates on the models were slightly perturbed before distribution for security reasons, which breaks some of the connectivity that would have otherwise been available. We assume that this is the main reason that the stripification algorithm finds only significantly shorter strips on this model. Finally, the stripification algorithm for ReduceM uses a greedy heuristic, which does not guarantee to produce good results in all cases since it does not optimize globally. In addition, many CAD models usually have a limit on how long strips can potentially be for any algorithm, which limits the impact of ReduceM. Finally, generating the ReduceM representation including stripification may be more computationally complex than comparable kd-tree or BVH construction algorithms, so more preprocessing time is required.

Model	Tris	Vertices	ReduceM fps	kd-tree fps
Powerplant	12.7M	11M	30.30	27.78
Double Eagle	82M	77M	4.61	5.95
Puget Sound	134M	67M	12.05	12.48
Boeing 777	364M	208M	4.76	n/a (OOM)
St.Matthew	372M	186M	4.81	n/a (OOM)

Table 3.4: **Results: Rendering performance.** *Performance results as average fps at* 1024^2 *pixels using* 2×2 *ray packets for primary visibility, compared to an optimized kd-tree implementation. Note that we do not report some numbers for the kd-tree since the data-set is out of core and thus fair comparison is impossible.*

Chapter 4

Deformable models

Unlike for static models, the largest problem in ray tracing of complex dynamic models is not memory, but maintenance of the acceleration structure. As hierarchies typically become invalid when geometric objects move or deform, the bottleneck in ray tracing shifts towards computing a new hierarchy after each deformation. In this chapter, we look at approaches for supporting ray tracing of complex deformable models, i.e. where geometry may move and deform but the number of objects stays the same. In particular, we show that bounding volume hierarchies are a better solution than kd-trees for this problem since they support a hierarchy refitting operation that is substantially faster than rebuilding the hierarchy. We also show how to improve the performance of interactive ray tracing using BVHs using ray packet techniques and other optimizations.

4.1 Deformable bounding volume hierarchies

A BVH is a tree of bounding volumes. Each inner node of the tree corresponds to a bounding volume (BV) containing its children and each leaf node consists of one or more primitives. Common choices for BVs include spheres, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs) or k-DOPs (discretely oriented polytopes). Many efficient algorithms have been proposed to compute sphere-trees [70], OBB-trees [51], and k-DOP-trees[84]. However, we use AABBs as the BV as they provide a good

balance between the tightness of fit and computation cost since efficient algorithms for ray-box intersection exist [132, 102, 159].

4.1.1 AABB hierarchies vs. kd-trees

In this section, we evaluate some features of BVHs based on AABBs and compare them with kd-trees for ray tracing. Most recent efficient and optimized ray tracing systems are based on kd-trees [149]. As far as static scenes are concerned, analysis has shown that optimized algorithms based on kd-trees will outperform BVH-based algorithms [61]. There are multiple reasons to explain this behavior: First, even the most optimized ray-AABB intersection test (e.g. from [159]) is more expensive than split plane intersection for kd-trees. This is due to the fact that in the worst case (i.e. no early rejection) up to 6 ray-plane intersections need to be computed for AABB trees, as opposed to just one for a kd-tree node. Another important aspect is that a BVH does not provide real front-to-back ordering during traversal. As a result, when a primitive intersects the ray, the algorithm cannot terminate (as is the case for a kd-tree), but needs to continue the traversal to find other intersections. Furthermore, kd-tree nodes can be stored more efficiently (8 bytes per node [151]) than an AABB possibly could. On the other hand, we found that BVHs often need fewer nodes overall to represent the scene as compared to a kd-tree (please see Table 4.3). This is mainly due to the fact that primitives are referenced only once in the hierarchy, whereas kd-trees usually have multiple references for each primitive since those overlapping a split plane need to be stored on both sides of the plane. In addition, AABBs have the advantage of providing a tighter fit to the geometric primitives with fewer levels in the tree, e.g. kd-trees need multiple subdivisions in order to discard empty space. Most importantly, the major benefit of BVHs is that the trees can be easily updated in linear time using incremental techniques. No similar algorithms are known for updating kd-trees.

4.1.2 BVH Construction

As was already described in chapter 2, we construct an AABB hierarchy in a top-down manner by recursively dividing an input set of primitive into two subsets until each subset has the predetermined number of primitives. We have found that subdividing until each leaf just contains one primitive yields the best results at the cost of a deeper hierarchy, as – similar to kd-trees – node intersection is cheaper than primitive intersection, although other authors have reported best performance for 6 primitives per node [101]. During hierarchy construction, the most important operation is to find a subdivision into two subsets that will optimize the performance of run-time ray hierarchy traversal. One of the best known heuristics for tree construction for ray tracing is the *surface-area heuristic* (SAH) [50, 61], which has been shown to yield higher ray tracing performance. However, despite recent improvements [146], it also has a much higher construction cost and will commonly take longer than the actual frame rendering time for dynamic environments. Because of this, we use the spatial median of one of the dimensions and sort the primitives into the child nodes depending on their location with respect to the midpoint. We observe that the spatial median build is usually about an order of magnitude faster to compute and provides rendering performance of 50-90% of SAH for most scenes. Note that even though we just split along one dimension, the bounding box will still be tight along all the three dimensions. As we are storing just one primitive per leaf, it is also easy to see that the total number of nodes in the tree for n primitives will always be $2n - 1$, which allows us to allocate the space needed for any sub-tree during construction.

Regardless of the heuristic for finding a split, the time complexity, $T(n)$,of the top-down AABB hierarchy construction algorithm is $\Omega(n \log_k n)$, where k is the number of children of each node and n the number of primitives. It is easy to see that for each split, every primitive in the node needs to be processed at least once to see which child it belongs in. Since at each level of the tree during construction all n primitives are

examined and the smallest possible number of levels is $\log_k n$, any top-down construction has to take at least $\Omega(n \log_k n)$ time.

4.1.3 Refitting the hierarchy

The main advantage of using BVHs for ray tracing is that animated or deforming primitives can be handled by updating the BVs associated with each node in the tree. Our algorithm makes no assumptions about the underlying motion or simulation. In order to efficiently update the hierarchy, we recursively update the BVHs by using a post-order traversal. We initially traverse down to the leaves from the root nodes. As we encounter a leaf node, we efficiently compute a new BV that has the tightest fit to the underlying deformed geometry. As we traverse from the leaf node in a bottom-up manner, we initialize the BV of an intermediate node with a BV of the leftmost node and expand it with the BVs of the rest of the sibling nodes.

The time complexity of this approach is $O(n)$, which is lower than the construction method. This is reflected by update times that we have found to be about 4 times faster than rebuilding the tree for our benchmark models (see Table 4.3 for detailed results). Therefore, we rely on hierarchy update operations to maintain interactive performance for dynamic environments.

4.1.4 BVHs for deformable scenes

We initially build an AABB tree of a given scene. As the model deforms or some objects in the scene undergo motion, the BVH needs to be updated or rebuilt. Updating the BVH is to recompute the bounds of each BV node, and rebuilding the BVH is to recompute the entire BVH from scratch and re-clustering the primitives. At run-time, we traverse the BVH to compute the intersections between the rays and the primitives.

If the algorithm only updates the BVH between successive frames, the run-time performance of BVHs can degrade over the animation sequence because the grouping of

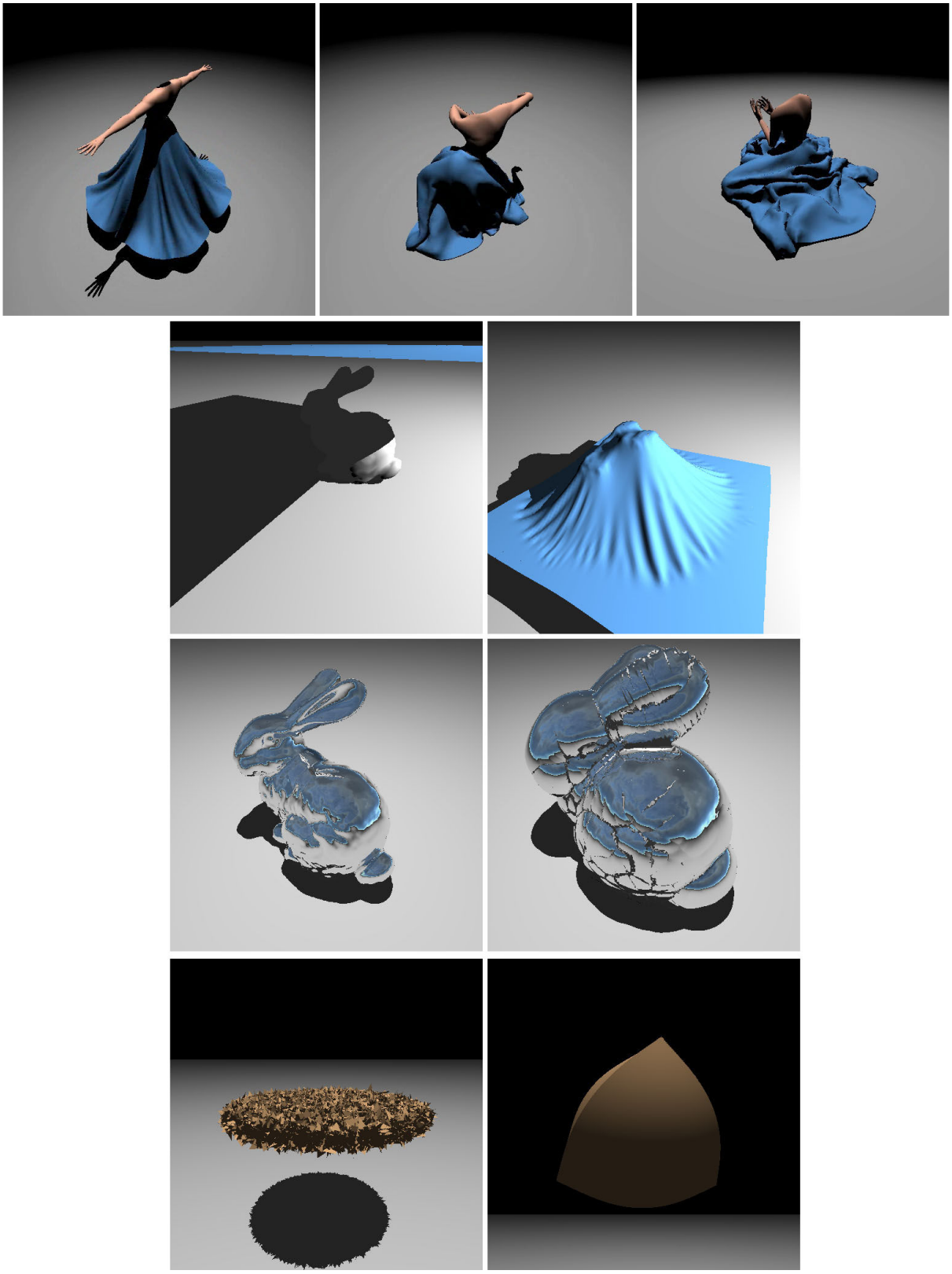


Figure 4.1: **Benchmark scenes:** *Frames from the dynamic benchmark models used for testing our BVH implementations, ranging from 16k to 69k triangles. From top to bottom: Princess, Cloth/Bunny, BART, Bunny. Please see Table 4.1 for more details.*

the primitives and structure of the hierarchy does not change. As a result, the BVs may not provide a tight fit to the underlying geometric primitives. This is often characterized by growing and increasingly overlapping BVs, which subsequently deteriorate the quality of the BVH for fast run-time BVH traversal by adding more intersections between the ray and AABBs. In such cases, rebuilding the AABB tree or parts of it is desirable.

We found that updating the BVH works well with relatively small changes to the scene or structured movement of groups of primitives such as meshes. When primitives move independently, however, for example in different directions, changes to the actual tree structure may be necessary to reflect the new positions of the deforming geometry. Still, rebuilding the BVH can be considerably more expensive than updating the BVH. As a result, we want to minimize the number of times rebuilding is performed. Therefore, we need to efficiently decide when updating the BVH is sufficient or rebuilding the BVH is required. This is non-trivial because the actual degradation of a BVH depends on many factors, such as the speed with which primitives move and the general characteristics of the motion of objects in the scene. Simple approaches such as rebuilding the tree every t frames have the disadvantage of not being adaptable to different characteristics over the animation and need to be chosen a priori. Conservatively choosing t means adding a lot of rebuilding overhead, which is especially unwanted in an interactive context. In order to efficiently detect when updating tree or rebuilding tree is required, we use a simple heuristic that is described in the next section.

4.1.5 Rebuilding criterion

We assume that BVH quality degradation is marked by bounding box growth that is not caused by actual primitive size, but by distribution of primitives or sub-trees in the box. For example, consider two primitives moving in opposite directions. The parent node containing them will have to grow to accommodate for the movement, resulting in a bounding box that is relatively large, but mostly empty. Since the probability that a

box will be intersected by a ray rises with its surface area, we want to rebuild a sub-tree to find a more advantageous tree topology. To find these cases and prevent them from impacting performance, we need to measure BVH degradation during each frame by using a simple and inexpensive heuristic.

Our heuristic is based on the idea that we can find nodes that are large relative to their children by comparing their surface area. In order to have a relative metric independent of scale, we measure the ratio of each parent node's surface area to the sum of the area of its two children. The larger the ratio becomes, the more imbalance exists in the sizes. We first compute the ratio during tree construction and store it in a field of the optimized AABB data structure (see next section.) Whenever the tree is updated, the changed surface areas are automatically computed and each inner node can easily calculate its new ratio. Since we assume that the ratio stored from the construction is as good as we can do, we find the difference between the new and old ratio and add them to a global accumulation value. Once the bottom-up update reaches the root, we have computed the sum of all the differences. To assure that this value can be tested independently of the tree size, we normalize it by dividing by the number of nodes that contribute to the sum, i.e. the sum of inner nodes, which is always $n - 1$ for n primitives. This yields a relative value describing the overhead incurred by updating the BVH instead of rebuilding it. This value is then simply compared to a predefined threshold value (we found a threshold of 40% to work well in our tests) and the tree is rebuilt if the threshold is exceeded.

This approach has several advantages: it will detect a good time to rebuild regardless of the actual frame rate and without any scene-specific settings. Furthermore, in scenes where there is little to no degradation, the heuristic will never need to initiate a rebuild. It is also possible to use the method to just rebuild sub-trees, but we found that this cannot fully replace a complete rebuild since degradations in the upper levels of the hierarchy typically have the highest impact on the performance of ray tracing. Therefore,

an implementation that rebuilds only sub-trees will have to either do a full rebuild sometimes, or support a top-level update where only the upper levels of the tree are rebuilt.

4.2 Fast ray tracing using BVHs

We now describe our BVH traversal algorithm and implementation and present techniques to extend the algorithm to multi-core architectures.

4.2.1 Traversal and Intersection with BVHs

We use a simple algorithm to compute the intersection of a ray and the scene primitives using the BVH. The ray is checked for intersections with the children of the current node starting at the root of the tree. If it intersects the child BV, the algorithm is applied recursively to that child, otherwise that child is discarded. Whenever a leaf node is reached, the ray is intersected with the primitives contained in that node. For most rays, the goal is to find the first hit point on the ray, so even if a ray-primitive intersection is found, the algorithm has to search the other sub-trees for potential intersections. An exception to this are shadow rays, where (at least for directional lights) any hit is considered sufficient and traversal can stop.

BVH traversal optimizations: Experience with kd-trees has shown that front-to-back ordering is a major advantage for ray tracing. Although BVHs do not provide a strict ordering, we found that storing the axis of maximum distance between children for each AABB and using that information during traversal together with the ray direction to determine a 'near' and 'far' child improves the traversal speed, especially for scenes with a high depth complexity (this has also been reported in [101] and [152]). Another issue is cache coherence during traversal: similar to the compact kd-tree representations

[153], we can optimize the AABB representation to fit within 32 bytes. We achieve this by storing the bounding box as 6 floating point values, one child pointer (such that the second child is expected to be directly after the first one in memory) and one float for storing quality information for the rebuild heuristic. Our profiling shows that BVH traversal using our AABBs has the same cache efficiency as the kd-tree traversal.

Use of ray coherence techniques: One of the main techniques used in current real-time ray tracers is to exploit ray coherence to reduce the number of traversal steps and primitive intersections per ray. Those algorithms were originally designed for the kd-tree acceleration structure. It is relatively straightforward to extend them to work with BVHs as well. In order to use coherent ray tracing [151] the BVH traversal has to be changed so that a node is traversed if *any* of the rays in the packet hits it and skipped if *all* of the rays miss it. A hit mask is maintained throughout the traversal to keep track of which rays have already hit an object and their distance. However, the traversal does no longer require that the rays have the same direction signs because unlike kd-trees the traversal order does not determine the correctness for a BVH. We have implemented ray packet traversal for 2×2 ray bundles using 4-wide SIMD instructions and found that it yields an overall speedup of about 2 to 3, which is even above the improvement obtained for kd-trees. We assume this is mainly because ray-AABB intersections are more costly than the kd-tree's ray-plane computation and therefore the reduction in traversal steps has a more pronounced effect on overall performance. Furthermore, we also support arbitrary-sized ray packets, which can be implemented very efficiently by using frustum culling such as presented in [124]. Depending on the detail level of the scene and the screen resolution, 16x16 or 8x8 packets will yield an even higher speedup to rendering and performs much better than the normal packet traversal code that tests each ray [101].

4.2.2 Multi-core architectures

One of major features of current computing trends is commodity architectures are becoming more parallel, mainly via multiple cores in one processor. Therefore, it is desirable to design our hierarchy update and run-time traversal such that they take advantage of available parallelism.

Hierarchy Update: Our update method can be parallelized relatively simply. Given the number of available threads, n, we decompose an input BVH into n sub-BVHs. For this, we simply compute n different children by traversing the tree from the root in the breadth-first manner. Then, each thread performs a bottom-up update from one of the computed nodes in parallel. After all the threads are done, we then sequentially update the upper portion of the n nodes. We particularly choose the bottom-up approach since it is well suited to parallel processing. For example, we do not require any expensive synchronization for each thread since the sub-trees that are accessed by threads are fully disjoint. Even though the BVHs computed for ray tracing are not balanced, this simple scheme provides reasonably good load balancing in practice. A more effective scheme might be to generate more sub-trees than processors and then use a work queue to assign trees to processors to make sure work is better balanced.

Overall, we have found that the main disadvantage of this approach is that it becomes limited by memory bandwidth relatively quickly and therefore may not scale very well with many cores. The post-order traversal operation is not very computationally intensive, but needs to read both the whole tree and every triangle, thus reading a potentially large amount of data. This is especially apparent with larger models where the sub-trees will not fit into processor caches.

Run-time traversal: We employ image-space partitioning to allocate coherent regions to each thread. Also, in order to achieve reasonably good load balancing, we first

Scene	Tris	Avg. fps	Update (ms)	Build (ms)
Bunny/Cloth	16K	13	4	13
BART	16K	11	6	23
Princess	40K	13	14	41
Bunny	69K	6	23	90
Buddha	1M	3	220	1659

Table 4.1: **BVH results** *Results for BVH ray tracing of several scenes. The benchmark configuration for each of the scenes is described in section 5. All benchmarks were performed at* 512^2 *resolution on a dual-core Intel Pentium 4 machine at 2.8 GHz using 8x8 ray packet traversal and secondary rays (shadows and reflections). Performance numbers are given as an average over the whole animation, tree build times are for the spatial median build.*

decompose image-space into small tiles (e.g. 16×16) and then allocate tiles to each thread. After a thread finishes its computation, it continues to process another tile. A more elaborate tile distribution may be necessary when using highly-parallel machines [138], but we have found that this approach works well for workstation-class machines and provides perfect scaling.

4.2.3 Performance results

We now show some results for our implementation of the algorithms above. We have implemented an interactive ray tracer for deformable models using BVHs running on a dual-core Intel Pentium 4 machine at 2.8 GHz. To compare the performance of BVHs with previous interactive ray tracing work for rendering static scenes, we also implemented kd-tree rendering (without animation capability). All acceleration structures support ray packet traversal using the SSE SIMD instruction set on Intel processors. For efficiency reasons, we only support triangles as primitives. We employ multi-threaded rendering and hierarchy updates using OpenMP.

We have tested our system on four animated scenes of varying complexity as well as one more complex static model to measure performance of our approach (see Fig. 4.1 and 4.2). In general, building a BVH tree using the naive midpoint method is

		kd-tree			BVH				skd-tree			
Scene	Tris	nodes	memory	build	nodes	memory	build	update	nodes	memory	build	update
Bunny/Cloth	16K	64137	859 KB	1487 ms	31923	997 KB	170 ms	4 ms	35097	548 KB	146 ms	4 ms
BART	16K	11075	1426 KB	1902 ms	32767	1024 KB	322 ms	6 ms	58921	920 KB	331 ms	11 ms
Princess	40K	218845	2778 KB	5 s	80059	2501 KB	733 ms	14 ms	148929	2327 KB	821 ms	19 ms
Bunny	69K	442347	5072 KB	10 s	138901	4340 KB	1526 ms	23 ms	259543	4055 KB	1521 ms	37 ms
Buddha	1M	2989439	33225 KB	80 s	2175431	67982 KB	32 s	220 ms	3666989	57296 KB	32 s	490 ms

Table 4.2: **Constructing hierarchical structures:** *Tree statistics for other acceleration structures as compared to BVHs. All hierarchies were built using the surface area heuristic instead of the spatial median and BVH build times are therefore higher than in the previous table (using the same machine). The SAH construction uses the simple $O(n \log^2 n)$ algorithm as opposed to the faster $O(n \log n)$ version [146]. Note that the memory requirements for skd-trees are only slightly smaller than for BVHs due to the higher number of nodes. Build times are about the same for both.*

more than one order of magnitude faster than the optimized surface-area heuristic kd-tree construction. In most cases, both structures have a similar memory footprint, but kd-trees need more nodes because references to primitives can be located in multiple nodes. The BART benchmark scene in particular is a hard case for hierarchy refitting methods since connectivity changes significantly during the animation. Our rebuild heuristic successfully detects these cases and only triggers a few rebuilds over the whole animation.

4.3 Comparing hierarchical structures

Recently, several acceleration structures were proposed that could be seen as a hybrid between BVHs and kd-trees. Because of BVH construction as a split operation, it is apparent that storing full bounding boxes may be redundant if a node essentially just stores the geometry as split along one dimension. Unlike kd-trees, which solve this by storing just the actual split coordinate and dimension, *spatial kd-trees* for ray tracing [163, 62] store two coordinates which represent the limits of the bounding boxes of the left and right child in the split dimension (which are allowed to overlap in case the contained geometry does). This reduces the memory requirements from storing 6 to 2

coordinates only. Similarly, Woop *et al.* [162] present a hardware implementation called the *b-kd-tree* in which they store the left and right bounds for both children and therefore use 4 coordinates per node. Construction for both structures is almost identical to BVHs by just storing the respective bounding box coordinates.

Both approaches also decrease the actual work done for one intersection as only 2 or 4 planes need to be intersected against the ray. In general, the traversal algorithm for spatial kd-trees is more similar to kd-trees with the difference that rays are now intersected against multiple planes. However, unlike kd-trees, no real depth sorting is provided, so traversal cannot stop after the first hit.

Implementation: To compare our BVH implementation against those approaches, we implemented a spatial kd-tree structure with two planes as in [163]. Similar to BVHs, we use 2×2 ray packets using SIMD instructions as well as packets of arbitrary size to allow direct comparison of results. For 2×2 packets, the traversal algorithm is a direct adaptation of ray packet traversal in kd-trees extended to test two planes. For larger packets, we designed a traversal algorithm that uses the inverse frustum culling described for kd-tree ray tracing in [124] for determining whether a packet intersects a node, although we do not perform entry-point search or split packets at the moment. For all three structures we constructed the hierarchy using a SAH build (as opposed to the spatial median build for BVH used in the results of the previous section).

Results and discussion: Our results are summarized in Table 4.3 and 4.2. In general, we observed an increase in overall rendering speed for static scenes when using skd-trees, which is a consequence of the less computationally expensive ray-node intersection. However, for animations one important disadvantage is that after the update, many of the empty leaf nodes introduced for empty space elimination may not be necessary any more or, even worse, would have to be used at a different point. We have found that this quickly results in more severe quality degradation of the hierarchy and, subsequently,

rendering requires very frequent rebuilds. To avoid this, we rebuilt the hierarchy every 5 frames for skd-trees, to allow a better performance comparison. We did not implement the B-kd-trees proposed in [162], but we assume that they would perform better than skd-trees in this regard since they would require less empty splits.

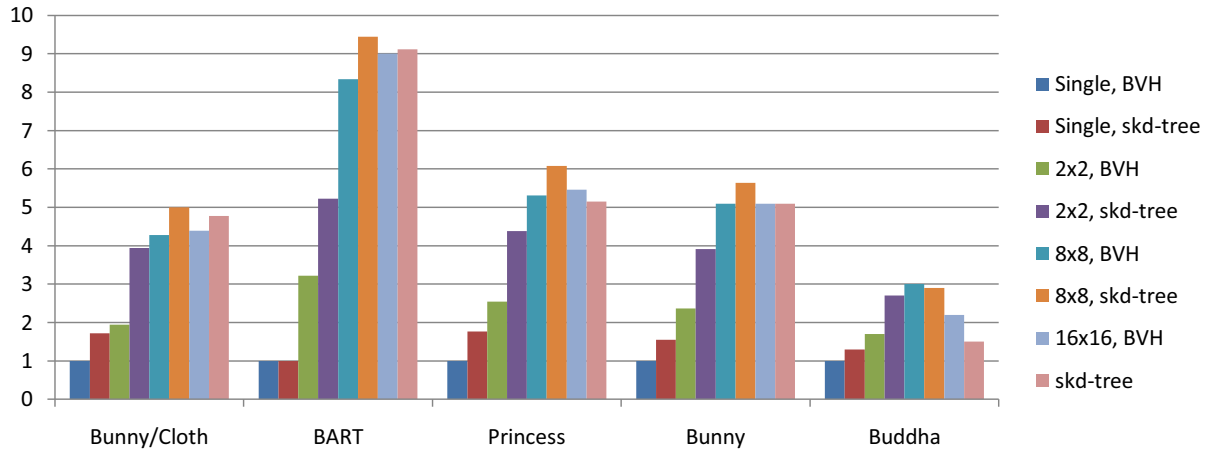


Table 4.3: **Rendering performance of BVH and spatial kd-tree:** *Direct comparison of rendering speed for BVH and our spatial kd-trees implementation. Benchmark results are frames per second relative to single ray BVH performance over the animation for* 1024^2 *primary rays only on a dual-core P4 machine at 2.8 GHz. The results are shown for different ray packet sizes and exclude all update times and rebuild times. In order to avoid excessive quality degradation when updating skd-trees, we rebuild the hierarchy every 5 frames.*

We also found that traversal for larger ray packets does not seem to scale up as well as for BVHs, so that skd-trees are about the same speed or even slower for our tested packet sizes. Although the individual nodes are only half as large as our AABB nodes (i.e. 16 bytes), memory use for skd-trees is only slightly lower than for BVHs. The reason for this is that in order to achieve good performance, extra splits to eliminate empty space at the outer bounds are needed often and, even though the actual empty leaves do not need to be stored, this increases the number of actual nodes in the tree. This also means that unlike BVHs, the actual number of nodes for a scene is not as predictable (although it can be bounded easily since only a limited number of empty space subdivisions can be introduced at each node), which prevents some easy ways

to optimize construction. Having more nodes also means that the tree is deeper and therefore on average more traversal steps may be needed to reach the leaves. Most importantly, though, the hierarchy update for animation is linear to the number of nodes. As the skd-tree usually has about twice as many nodes, this means that updating may take longer.

Finally, a subtle difference is that ray packet traversal for skd-trees in general can be more complicated to implement and less versatile: as for kd-trees, groups of rays for inverse frustum culling are limited to having the same direction signs, which can make obtaining groups of coherent rays more challenging, in particular for secondary rays. Adapting an omnidirectional traversal for skd-trees may alleviate this problem [123]. In contrast, BVH traversal is independent of ray direction signs, which eliminates special cases for traversal, and frustum culling can be introduced easily by the fast frustum-box intersection described in [124].

In conclusion, our results suggest that for animated scenes with updates a BVH implementation is to be preferred and also lends itself better to an optimized ray packet implementation. For scenes with varying numbers of primitives, the hierarchy update will not work, so in that case a fast rebuild such as described in [163] should be used instead. For static scenes, standard kd-trees will very likely provide superior performance with an MLRT implementation [124], albeit at higher memory cost and more complex optimized hierarchy construction.

Chapter 5

Fast parallel hierarchy construction

Hierarchy refitting approaches such as presented in the previous chapter are generally the fastest solution to hierarchy maintenance. However, their main shortcomings are that they do not support more general dynamic scenes where the number of objects may change and that they will still require a rebuild operation in case hierarchy quality degradation is detected.

This chapter investigates fast methods for constructing and refitting object hierarchies such as BVHs. In particular, the focus in the approach presented here is to utilize highly-parallel architectures such as GPUs to speed up the construction process significantly and to allow interactive building of ray tracing optimized BVHs.

In recent years, the focus in processor architectures has shifted from increasing clock rate to increasing parallelism. Some of the most successful examples are GPUs which have become very general processors with a strong focus on achieving performance through high parallelism. Current high-end GPUs have a theoretical peak speed of up to a few Tera-FLOPs, thus far outpacing current CPU architectures. Specifically, there are several features that distinguish GPU architectures from multi-core CPU systems and also make it harder to achieve peak performance. First, the GPUs usually have a high number of independent cores (e.g. the current generation NVIDIA GTX 280 has 30 cores) and each of the individual cores is a vector processor capable of performing

the same operation on several elements simultaneously (e.g. 32 or 64 in current GPUs). Second, GPUs only provide no or a very limited general cache hierarchy for all memory accesses, unlike CPUs. Instead, each core can handle several separate tasks in parallel and switch between them in hardware when one of them is waiting for a memory operation to complete. This hardware multi-threading approach is thus designed to hide the latency of memory accesses to perform some other work in the meantime. However, both of these characteristics imply that – unlike CPUs – achieving high performance in a GPU-based algorithm critically depends on not only providing a sufficient number of parallel tasks so that all the cores are utilized, but providing several times that number of tasks just so that cores have enough work to perform while waiting for data from relatively slow memory accesses. This greatly affects the algorithm design for the GPU architectures.

5.1 SAH hierarchy construction

The surface area heuristic [50, 100, 61] method for building hierarchies can be applied to many types of hierarchical acceleration structures – among them object and spatial hierarchies – since it has been shown to be a good indicator of expected ray intersections. The time complexity for top-down construction of a hierarchy for n triangles is $O(n \log n)$ due to the equivalence to sorting, and research has shown [147] that SAH optimized construction can also be achieved in the same bound. Top-down builders proceed by recursively splitting the set of geometric primitives — usually into two parts per step, resulting in a binary tree. The manner in which the split is performed can have a large impact on the performance of a ray tracer or other application using the BVH. The SAH provides a local cost function that allows to evaluate all possible split positions at a node and then pick the one with the lowest cost.

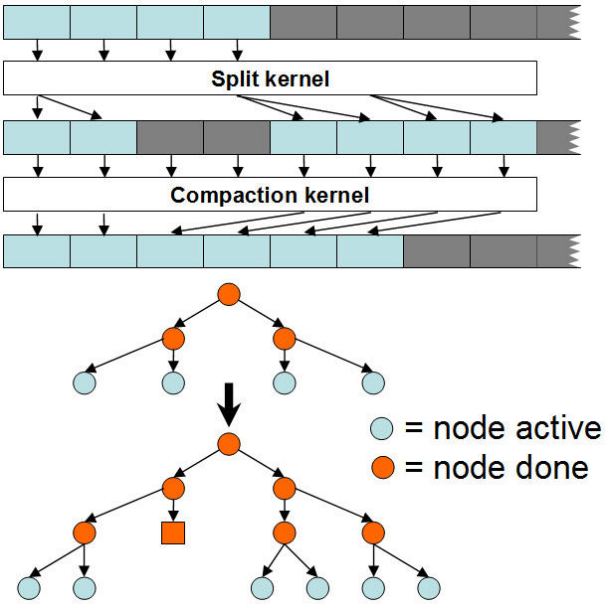


Figure 5.1: **Work queue construction:** *By using two work queues, we can run all active splits in parallel. Since the split kernel may not output new splits, a compaction step after each split level removes empty space in the queue and allows it to be used in the next step.*

5.2 GPU SAH construction

The main challenge for a GPU algorithm for hierarchy construction is that it has to be sufficiently parallel so as to exploit the computational power of the processors. In a top-down construction algorithm, there are two main ways to introduce parallelism: a) process multiple splits in parallel and b) parallelize an actual split operation. In our approach, we use multiple cores to run as many node splits in parallel as possible and then use the data parallel computation units in the processors to accelerate the SAH evaluation as well as the actual split operation.

5.2.1 Breadth-first construction using work queues

We now present our algorithm for parallelizing the individual splits across all the cores. During top-down construction, each split results in new nodes that still need to be processed, i.e. split as well. Since each of the splits can be processed totally independently,

processing all the open splits in parallel on multiple cores is an easy way to speed up construction. The simplest approach to manage all this work is to introduce a queue that stores all the open splits such that each processor can fetch work whenever it is finished working on a node, as well as put new splits on the work queue when finishing up. However, even though current GPU architectures support atomic operations that would make global synchronization possible, the problem is that the same architectures do not provide a memory consistency model that would ensure writes to the queue are seen in the correct order by other processors. Thus, implementing a global work queue may not be practical yet (or desirable for performance reasons, see chapter 6.2 for more discussion.) Instead, we opt for an iterative approach that processes all the currently open splits in parallel and writes out new splits, but then performs a queue maintenance step in between to provide a valid queue for the next step (similar in spirit to [68].)

In a binary tree, each split can result in either 0, 1 or 2 new open splits as a result. Thus, if we have m open splits, we only need a work queue of at most size $2m$ to hold all potential new splits. By using two work queues, one as an input and one as an output, we can perform parallel splits without any explicit coordination between them. In our algorithm, each block i reads in its split item from the input queue and processes it (as described in the next section) and then either writes out new splits or a null split to the output queue at positions $2i$ and $2i + 1$ (also see Fig. 5.1). After that step, the output work queue may have several null splits that need to be eliminated. Therefore, we run a compaction kernel (such as described in [128]) on the queue and write the result into the input queue. We again run the split algorithm on the new input queue as long as there are still active splits left. Note that this means that we generate and process all the nodes in the tree in breadth-first order, whereas recursive or stack-based CPU approaches typically proceed in depth-first order (Fig. 5.1). Note that this form of work organization does not add work overhead and the the overall work complexity is still $O(n \log n)$.

5.2.2 Data-Parallel SAH split

In general, performing a SAH split for an object hierarchy consists of two steps:

1. Determine the best split position by evaluating the SAH.

2. Reorder the primitives (or the indices to the primitives) such that the order in the global list corresponds to the new split.

Note that unlike spatial partitioning approaches such as kd-trees each primitive can only be referenced on one side of the split (commonly determined by the location of the centroid of the primitive), which means that the reordering can be performed in-place and no additional memory is needed. All of these steps can be performed by exploiting data parallelism and using common parallel operations such as reductions and prefix sums, and each split is executed on one core.

We perform approximate SAH computation as described in [71, 116] and generate k uniformly sampled split candidates for each of the three axes, and then use $3k$ threads so that we test all the samples in parallel. Note that when k is larger or equal to the number of primitives in the split, we can instead use the positions of the primitives as the split candidates, in which case the algorithm produces exactly the same result as the full SAH evaluation. The algorithm loads each primitive in the split in turn while each thread tests its position in reference to its split position and updates the SAH information (such as bounding boxes for the left and right node) accordingly. Once all primitives have been tested, each thread computes the SAH cost for its split candidate. Finally, a parallel reduction using the minimum operator finds the split candidate sample with the lowest cost. We also test the SAH cost for not splitting the current node here and compare it to the computed minimal split cost. If it is lower, then we abort the split operation and make the node a leaf in the hierarchy.

Given the split coordinate as determined by the previous step, our algorithm then needs to sort the primitives into either the left or the right child node. To avoid copying

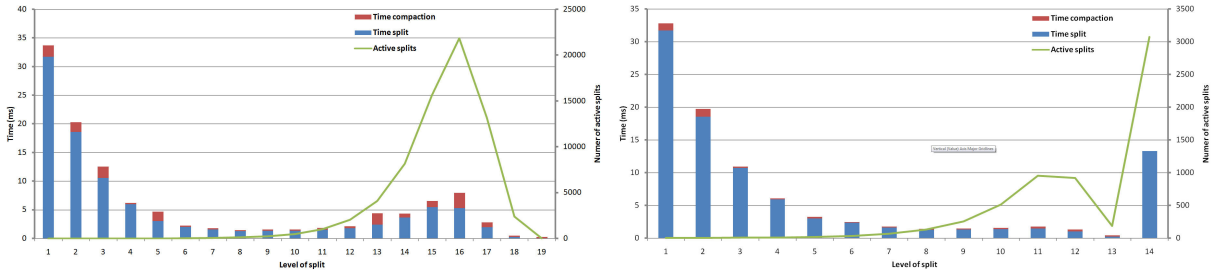


Figure 5.2: **Construction timings per level:** *In order to evaluate our SAH construction algorithm, we show the time taken to run the split as well as compaction kernels for the 69K Bunny model per level of the tree, starting with the one split on level 1. We also show how many splits are active at any level. Left: normal construction algorithm. Right: construction with small split kernel as described in the text.*

the geometry information, we only reorder the indices referencing the primitives instead. To parallelize this reordering step, we go over all the indices in chunks of a size equal to the number of threads in the block. For each chunk, each thread reads in the index and set a flag to 1 if the respective primitive is to the left or 0 if on the right. We then perform a parallel prefix sum over the flags. Based on that, each thread can determine the new position to store its index. Finally, we store the bounding box information computed during SAH computation in both the child nodes and save back the new split information to the output split list for the next step if there is more than one primitive left in the respective node.

5.2.3 Small split optimizations

The algorithm as described above generates a hierarchy practically identical in quality to CPU-based approximate SAH techniques (see the results section). In practice, we can efficiently use more samples due to high data parallelism and the SAH quality may be even better since the main change has been to reorganize the split operation such that it uses parallelism at every stage. Fig. 5.2 a) illustrates the performance characteristics of the algorithm by showing the time spent during construction by level of the split. There are two main bottlenecks in the algorithm: similar to CPU implementations, the initial

splits at the top levels of the hierarchy are slow due to lack of processor parallelism and large numbers of very small splits at the end. All further splits are much faster even though there is much more work done per level because computational resources are more fully utilized and processors can hide memory latency by switching between active tasks. We present an improved scheme to improve the parallelism at the top levels in section 5.3, but we can also improve on the performance at the lower levels of the hierarchy. The main sources of overhead here are a) higher compaction costs since a high number of splits are generated during each step and b) low vector utilization due to only processing a few primitives per split. There is also some constant administrative overhead during each operation for reading in information about the split etc. that becomes more significant as the actual computational intensity decreases. We address these issues by using a different split kernel for all splits with sizes below a specified threshold and modify the compaction kernel such that these splits are automatically filtered from the main work queue into a small split queue. Once all the normal splits are done, we can run the small split kernel on all the elements in that queue once.

The main idea about the new split kernel is to use each processor's local memory to maintain a local work queue for all the splits in the sub-tree defined by the input split, as well as to keep all the geometric primitives in the sub-tree in local memory as well. Since local memory is very fast, we can then run the complete sub-tree's construction entirely without waiting for memory accesses. Essentially, we are using this memory as an explicitly managed cache to reduce the memory bandwidth needed for construction. In addition, we use as few threads as possible in the kernel to maximize utilization of the vector operations. The threshold value of primitives for a split to be "small" depends on how much geometry data can be fit into local memory, i.e. the cache size. In our case, we have set both the thread count as well as the threshold to 32, which for our GPU architecture is the effective SIMD width and will also fill up most of local memory. Fig. 5.2 b) shows the impact of introducing this split kernel. As the normal construction is

run, all splits smaller than the threshold are removed from the main work queue, so the number of splits falls much more quickly than previously. At the very last split step, the small split kernel is invoked and completely processes all the remaining sub-trees in one run. Overall, we have found that this leads to a 15-20% overall speedup in our test cases due to practically full utilization of the computational resources and reduced overhead for queue compaction.

5.3 Hybrid construction algorithm

Fig. 5.2 shows that a large fraction of the overall construction time is spent in the top levels of the tree. Similar to findings in multi-core CPU construction [116], this is because having only one or very few active splits such as at the beginning of the construction means that only very few of the overall processor cores are active. In addition, these splits also are the most expensive ones since they contain the most triangles. Overall, the SAH construction method above therefore does not scale well to larger models since the performance will be dominated by the early, almost serial construction stages.

However, it is also possible to combine the algorithm described here with another that works efficiently for large numbers of triangles. The LBVH algorithm as presented in [94] is a GPU algorithm that reduces the hierarchy construction problem to a sort on a space-filling curve. Using a fast parallel sort implementation [128], it is very efficient on GPUs but has the main disadvantage that it does not take hierarchy quality for ray tracing into account. In this context, on the other hand, it gives us the possibility to use only a subset of the bits in the Morton code to sort on. In this case, the LBVH builds a very shallow hierarchy with large numbers of primitives at the leafs. We can then treat all the leafs in that hierarchy as active splits and process them in parallel with the SAH construction algorithm. Essentially, this is similar to the bounding interval hierarchy [163] (BIH) and grid-assisted [150] build methods, but in a highly parallel

Model	Tris	CPU SAH	GPU SAH	LBVH	Hybrid	Parallel SAH
Flamenco	49K	144ms 30fps/99%	85ms 30.3fps/100%	9.8ms 12.4fps/41%	17ms 29.9fps/99%	*n/a*
Sibenik	82K	231ms 21.4fps/97%	144ms 21.7fps/98%	10ms 3.5fps/16%	30ms 21.4fps/97%	*n/a*
Fairy	174K	661ms 11.5fps/98%	488ms 21.7fps/100%	10.3ms 1.8fps/15%	124ms 11.6fps/99%	21ms 93%
Bunny/Dragon	252K	842ms 7.8fps/100%	403ms 7.75fps/100%	17ms 7.3fps/94%	66ms 7.6fps/98%	20ms 98%
Conference	284K	819ms 24.4fps/91%	477ms 24.5fps/91%	19ms 6.7fps/25%	105ms 22.9fps/85%	26ms 86%
Soda Hall	1.5M	6176ms 20.8fps/98%	2390ms 21.4fps/101%	66ms 3fps/14%	445ms 20.7fps/98%	*n/a*

Table 5.1: **Construction timings and hierarchy quality:** *First row for each scene: Timings (in ms) for complete hierarchy construction. Second row: relative and absolute ray tracing performance (in fps) on our GPU ray tracer compared to full SAH solution at* 1024^2 *resolution and primary visibility only. CPU SAH is our non-optimized approximate SAH implementation using just one core, GPU SAH is the algorithm as presented in section 5.2 and Hybrid the combined algorithm as presented in section 5.3. The parallel and full SAH are both from the grid-BVH build in [150] and were generated on 8 and 1 core of an Intel Xeon system at 2.6 GHz, respectively.*

approach. Overall, as we will show in the result section, this combines the speed and scalability of the LBVH algorithm with the ray tracing performance optimizations in the SAH algorithm.

5.4 Results

We now highlight results from the algorithms described in the previous sections on several benchmark cases and scenarios. All algorithms are run on a Intel Xeon X5355 system at 2.66GHz running Microsoft Windows XP with a NVIDIA Geforce 280 GTX graphics card with 1 GB of memory and were implemented using the NVIDIA CUDA programming language [109]. We use several benchmark scenes chosen to both allow comparison to other published approaches as well as to cover several different model characteristics such as architectural and CAD scenes as well as scanned models (see Fig. 5.3). Note that all benchmark timings cover the full construction process starting with

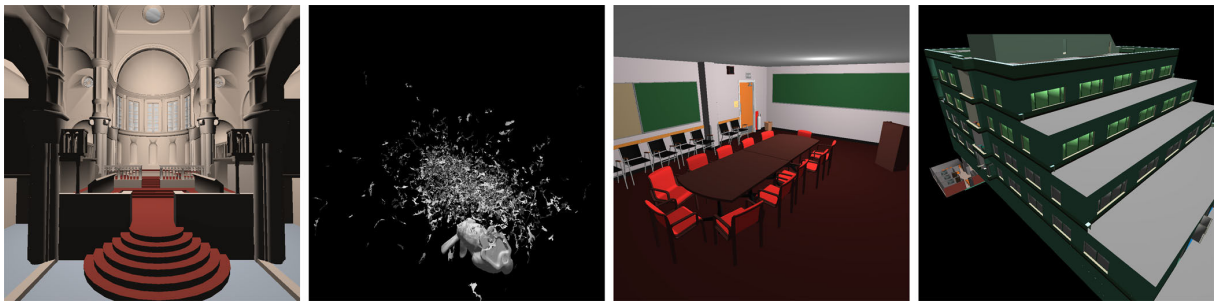


Figure 5.3: **Benchmark models:** *Our benchmark scenes used to generate results. From left to right: Sibenik cathedral (80K tris), Bunny/Dragon animation (252K tris), Conference room (284K tris), Soda Hall (2M tris).*

building the initial bounding boxes, but do not include any CPU-GPU copy of geometry. In our benchmark results for the hybrid algorithm, we used the LBVH algorithm for performing the first 6 levels of splits before switching to SAH construction. All performance results and images in the paper were produced with a BVH-based ray tracer running on the GPU. We implemented a relatively simple and unoptimized packet-based BVH ray tracer similar to [58] in CUDA and tested it on our benchmark scenes while rebuilding the hierarchy for each frame. In combination with the hierarchy construction, this allows interactive ray tracing of arbitrary dynamic scenes on the GPU.

To justify both the approximate SAH building algorithm as well as the hybrid approach, we first compare relative rendering speed of all the algorithms presented in this paper to a hierarchy built on the CPU with a full SAH construction algorithm such as described in [152]. Rendering is performed using a standard CPU BVH ray tracer using ray packets [93, 152]. The results listed in table 5.1 demonstrate that for most scenes using approximate SAH has close to no impact compared to a full SAH build and that even the hybrid build is very close to the reference results. This backs results of similar splitting approaches in [163, 150]. Note that in some cases the approximate or hybrid algorithm is in fact faster than the reference implementation. The SAH is only a local heuristic and thus it is possible that a split that is non-optimal in the SAH sense may still result in better performance in the actual ray tracer. The efficiency of

the LBVH construction is highly scene dependent. It can provide adequate quality for scenes with evenly distributed geometry such as the Dragon/Bunny model, but for architectural and CAD scenes the performance is clearly inferior. Because it uses a fixed number of bits per axis, its quality can degrade when very small details exist in a large environment (the classic *teapot-in-stadium* scenario). Thus, the Fairy scene (benchmark 3 in Table 5.1) with a complex model in the middle of a less complex environment is an example of this problem. In the hybrid algorithm, it is always possible to limit the impact by limiting the number of splits performed by LBVH. Furthermore, this problem is mitigated if the scene has some pre-existing structure, say from a scene graph that can assist the construction procedure. Having primitives with highly varying sizes can also have an impact on performance here since the classification according to Morton code does not take this into account. However, this is usually also a problem for other construction algorithms and the common solution would be to subdivide large primitives [37] before construction. Table 5.1 also shows timings for the actual construction with our different approaches and also lists published numbers from a fast parallel CPU BVH builder [150] as well as our non-parallel CPU implementation of a sampling based SAH approach (similar to [71, 150]) using 8 samples/split. Note that since the GPU ray tracer uses a higher number of samples (in our case 64), it can provide slightly better hierarchy quality compared to the CPU solution. The full LBVH is the fastest builder, but is dominated by kernel call and other overhead for small models.

5.4.1 Analysis

Current GPU architectures have several features that would make them suitable for hierarchy construction. First, thanks to special graphics memory they have significantly higher memory bandwidth. At the same time, even though the individual processing units do not have a general cache, they have explicitly managed fast local memory. Thus, if data for the current computation can be loaded from main memory and held in

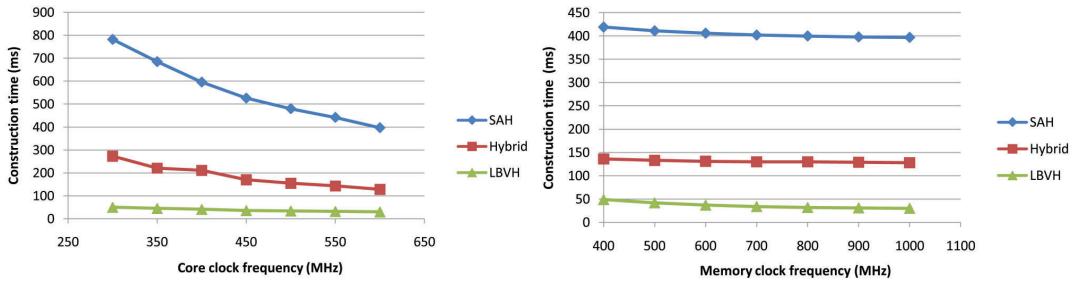


Figure 5.4: **Bottleneck analysis:** *Relative performance of the three construction algorithms when modifying the core processor clock (top) and the memory clock (bottom).*

local memory, memory access and bandwidth are very fast. GPUs also have very high available parallelism both in terms of independent processors as well as data parallel units. However, exploiting this parallelism in the construction algorithm now becomes an harder challenge. Unlike the thread parallelism in CPU implementations, using data parallelism is both more important as well as difficult due to higher SIMD width.

The memory footprint of our construction algorithm is relatively low since object hierarchy construction does not require temporary memory and all elements can just be reordered in-place. We can size the main work queues conservatively since for n primitives there can be at most $n/2$ splits active at a time. Additionaly, the algorithm stores an AABB per primitive, the main index list as well as an array for holding the BVH nodes (which can be conservatively sized at the theoretical maximum of $2n-1$ nodes.) Overall, our algorithm uses up to 113 bytes/triangle during construction including the BVH, assuming the worst case of one triangle per leaf. This allows construction of hierarchies for multi-million triangle models on current GPUs without problems.

We also analyzed the current bottlenecks in our construction algorithms. Figure 5.4 shows the impact of independently changing core (i.e. processing) and memory clock for our construction methods. It is obvious that all three are currently bound by computation, and both the hybrid and SAH algorithms are not limited by memory bandwidth. The LBVH algorithm shows the highest dependence on memory speed: since GPU sort is known [52] to be bandwidth limited and construction must have the

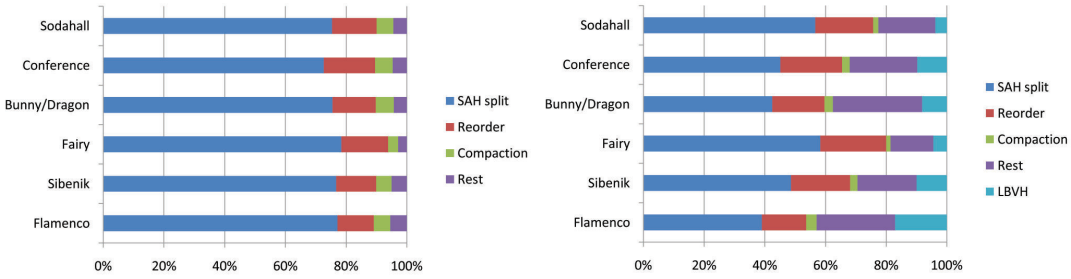


Figure 5.5: **Time spent:** *Split-up of the total time spent in construction into each part of the algorithm. Left: Full SAH build. Right: Hybrid build. The "rest" times consist of all operations not explicitly in other parts, such as reading and writing BVH nodes and setting up all the other components.*

same lower bounds as sorting, we take this as a sign that the LBVH build is close to the limit of achievable scalability. Overall, this means that the construction algorithms will very likely scale well with added computational power in future architectures.

We also analyzed where time is spent in the construction. Figure 5.5 shows a break-up of relative time for each of our benchmark scenes into several components: *SAH split* is the time for finding the optimal subdivision, *Reorder* is the time spent reordering the triangles according to the split, *Compaction* consists of the time for compacting and maintaining the work queues between splits and *LBVH* is the initial split time in the hybrid construction (for 6 levels). The remaining time (*Rest*) is spent reading in and writing back BVH node information, setting up splits and otherwise joining the rest of the steps together. Note that we did not include the time for computing the AABBs and Morton codes here as it makes up far less than 1% of the total time in all benchmarks. Overall, the results show that the full SAH build is clearly dominated by the cost of evaluating the SAH split information, while the hybrid build is more balanced. Computing the initial splits using the LBVH algorithm only takes a relatively small amount of time, but reduces both the overall construction time as well as some of the cost in work queue management.

5.4.2 Comparison

There has been some concurrent work on GPU hierarchy construction focused on building kd-trees [166]. Our approach is similar in the organization of the splits by using a work queue, but the authors use spatial median splits after sorting primitives into chunks in order to speed up the top-level split computation. Despite the similarities in both approaches, we would like to point out some differences in BVH and kd-tree construction. In particular, the memory overhead in object hierarchy construction is very small, whereas dynamic allocation in kd-tree construction so far limits the GPU implementation — the current kd-tree algorithm would need about 1GB of memory to construct the kd-tree for a model with one million triangles, whereas we could support about ten times that. On the other hand, BVH splits are intrinsically more computationally intensive as the split kernel has to compute bounding boxes for each split candidate whereas kd-tree splits are directly given by the parent's bounding box and the split plane. Thus, the main inner loop needs to load 3 times the data (all 6 bounds as opposed to 2) with a corresponding increase in computation. This difference may explain that our construction times are slightly higher than the kd-tree numbers in [166] for the Fairy model (77ms compared to 124ms in our hybrid approach.)

Chapter 6

Applications

The previous chapters of this thesis have introduced several methods for ray tracing on complex static and dynamic models with a focus on visualization. In this chapter, we demonstrate several applications of previously discussed algorithms to other areas and show how they can improve the respective state-of-the-art.

6.1 Interactive sound simulation using frustum tracing

In this section we present an algorithm for interactive sound propagation in complex and dynamic scenes that builds on the BVH algorithms presented in chapter 4. Our approach uses a simple volumetric representation based on a four-sided convex frustum, for which we describe efficient algorithms to perform hierarchy traversal, intersection and specular reflection and transmission interactions at the geometric primitives. Unlike beam tracing and pyramid tracing algorithms, we perform approximate clipping by using a subdivision into sub-frusta. As a result our rendering algorithm reduces to tracing ray packets and maps well to the SIMD instructions available on current CPUs. Compared to state-of-the-art beam tracing implementations, this approach is far more scalable to complex models and performs significantly faster, allowing interactive performance on

multi-core CPU systems.

6.1.1 Sound propagation in virtual scenes

Sound propagation is the process of simulating the path of sound waves in a virtual scene with one or multiple sound sources (usually modeled as points) as well as receivers (or listener). The sound waves can be reflected, transmitted or diffracted by objects in the scene and may at some point reach the receiver. The superposition of all these waves then is what the listener hears.

Sound propagation approaches can be broadly categorized into numerical and geometric simulation methods. Numerical solutions [88] attempt to accurately model the propagation of sound waves by numerically solving the wave equation. These methods are general and highly accurate [111]. However, they can be very compute and storage intensive [143]. Current approaches are too slow for interactive sound propagation in complex environments and are mainly limited to extremely simple scenes. Geometric solutions model the propagation of sound based on rectilinear propagation of waves and can, while less accurately, model the early reflections in sound propagation. Most of these methods are closely related to parallel techniques in global illumination, and many advances in either field can also be applied to the other. The earliest of these approaches were particle and ray based [86, 89] and simulated the propagation paths by stochastically sampling them using rays. However except for a very high number of samples, these techniques suffer from noise and aliasing problems [97], both spatially and temporally. Approaches using discrete particle representations called *phonons* or *sonels* [12, 26, 79] have been developed in the last few years. While promising, they are currently limited to simple scenes. Moreover, particle and ray-based algorithms are susceptible to aliasing errors and may need a very high density of samples to overcome those problems. *Image source* algorithms create virtual sources for specular reflection from the scene geometry and can be combined with diffuse reflections and diffractions

[13, 25]. They accurately compute the propagation paths from the source to the listener, but the number of virtual sources can increase exponentially for complex scenes [13]. This makes these techniques suitable only for static scenes.

The last type of geometric methods is based on *beam tracing*, which recursively traces pyramidal polyhedra from the source to the listener [63, 32, 39]. Funkhouser et al. [43, 42] showed how beam tracing methods can be used for sound propagation at interactive rates in complex virtual environments. Some algorithms have been proposed to use beam tracing on moving sources [5, 45]. However, current algorithms take large pre-processing time, are problematic on scenes that are not densely occluded and are not directly applicable to dynamic scenes with moving objects.

6.1.2 Frusta for sound propagation

In order to avoid the sampling issues but retain the advantages of beam tracing, we trace a simple convex polyhedron instead of infinitesimal rays. Specifically, we perform *frustum tracing* which is similar to beam tracing and pyramid tracing. We use a simple convex frustum so that we can perform fast intersection tests with the nodes of the hierarchy and the primitives. Unlike beam tracing algorithms, we perform approximate clipping using ray packets. This amounts to a discrete approximation to the arbitrary beam shape achievable in beam tracing. Overall, our representation combines many of the speed advantages of ray packet tracing with the benefits of volumetric formulations.

We use a convex four-sided frustum, i.e. a pyramid with a quadrilateral base (see Fig. 6.1(a)) that is defined by its four side faces and one front face. Equivalently, the frustum can be represented as the convex combination of four corner rays defining the frustum. At a broad level, the main difference between frustum and beam tracing is how we keep track of intersections with the primitive and the scene. Beam tracing performs exact clipping with each primitive in the scene and therefore needs to maintain a full list of clipped edges or faces of the beam. We avoid these relatively expensive operations by

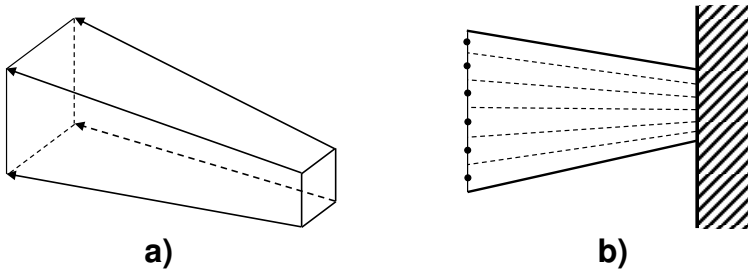


Figure 6.1: **Frustum-based packet:** *The frustum primitive used in our algorithm. a) The frustum is defined by the four side faces and the front face, or equivalently by the boundary rays on the sides where the faces intersect. b) the frustum is uniformly subdivided into sub-frusta defined by their center sample rays (dots), depending on a sampling factor.*

subdividing the frustum uniformly into smaller sub-frusta to perform discrete clipping, and only keep track of intersections at the level of those sub-frusta (see Fig. 6.1(b)). Moreover, each sub-frustum is represented by a sample ray, and a sub-frusta is considered to intersect a primitive only if its sample ray hits the primitive. Essentially, this can be interpreted as a discrete version of a clipping algorithm and can introduce some errors in our propagation algorithm.

The difference between the frustum and beam tracing process is also highlighted in Fig. 6.2. We show the intersection of the beam (left) and frusta (right) with three primitives and the resulting secondary beams and frusta computed for reflection and transmission. Note that since the intersection is determined by the location of the sample ray, the frustum tracing algorithm in this example will underestimate the size of secondary beams at the primitive on the left. The amount of error introduced depends on the sampling rate, i.e. the rate of subdivision of the frustum.

Benefits: Our formulation of the frustum and the clipping algorithm allows a faster and more general algorithm for propagation. We use the main frustum as a placeholder for all the enclosed sub-frusta during hierarchy traversal or intersection computations. As a result we are able to achieve very efficient and fast traversal using our representation in both static and dynamic scenes. In addition, we organize our sample rays in ray pack-

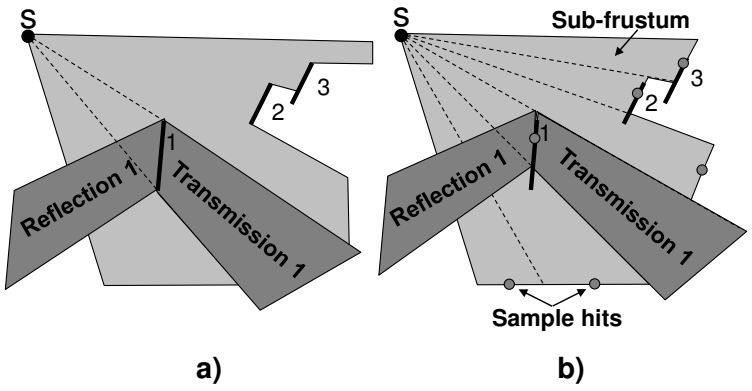


Figure 6.2: **Beam vs. frustum tracing:** *Our approach compared to beam tracing for a simple example.(Left): beam tracing. (Right): frustum tracing. The discrete sampling in our frustum based approach underestimates the size of the exact reflection and transmission frustum for primitive 1 and overestimates the size for primitives 2 and 3.*

ets similar to those used in interactive ray tracing, and exploit the uniform subdivision of frusta for faster primitive intersection computations. Finally, we defer constructing the actual sample ray computation until the sub-frusta are actually needed, i.e. if the whole frustum does not fully hit a primitive. This reduces the set-up cost, especially for very small beams.

6.1.3 Frustum Tracing

The goal of frustum tracing is to identify the primitives (i.e. triangles) that intersect the frustum and then to construct new secondary beams that represent specular reflection and transmission of sound. This involves traversing the scene hierarchy, computing the intersection with primitives and then constructing secondary frusta. We present algorithms for each of these computations.

Construction of secondary frusta: Whenever a frustum hits a primitive, we construct secondary frusta for transmission and specular reflection. If the entire frustum hits one primitive, the construction of the secondary frusta is simple and can be accomplished by just using the four corner rays. For the general case, when different sub-frusta hit different primitives, multiple secondary frusta have to be generated. A naïve solution

would be to generate reflection and transmission for each single sub-frustum defined by a sample ray. However, this could result in an extremely high number of additional frusta, and the complexity of the algorithm will grow as an exponential function of the number of reflections. To avoid this, we combine those sub-frusta that hit the same primitive by hierarchically comparing four neighboring samples and treating them as one larger frustum (see Fig. 6.3). This can be seen as a quad-tree structure, although we do not compute the tree explicitly. If the samples hit neighboring primitives that have the same material and normal, we combine those primitives in the same way to avoid splitting too many sub-frusta. This is especially useful when rectangles are represented by two triangles, which is a common case in architectural models. In practice, we have found that our approach yields a good compromise between the time taken to find optimal groups of sub-frusta and the number of secondary frusta needed. We also exploit the fact that the combined frustum exactly represents the sub-frusta, and there is no loss of accuracy due to this hierarchical grouping. If the primitives in the scene are over-tessellated, we could use simplificationalgorithms to decrease their size [77]. This can introduce some additional error in our propagation algorithm, but big triangles in the scene would result in fewer secondary sub-frusta.

Hierarchy traversal: We use a bounding volume hierarchy (BVH) as our choice of scene hierarchy, as it has been shown to work well for general dynamic scenes as discussed in earlier chapters. However, our algorithm can also be adapted to be used with kd-trees or other hierarchies. The main operation for traversal of the BVH is checking for intersection with a BV, most commonly an axis-aligned bounding box (AABB). As described by Reshetov et al. [124], a frustum can be tested for overlap with an AABB quickly. If the frustum does not intersect the AABB node, the entire subtree rooted at that node can be culled. Otherwise the children of the node are tested in a recursive manner.

However, this traversal method can result in traversing too many nodes because

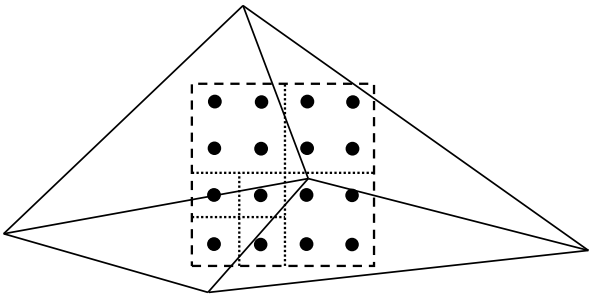


Figure 6.3: **Constructing secondary frusta:** *We compute reflected and transmitted frusta efficiently by grouping sub-frusta that hit the same primitive together in a single secondary frustum instead of having to trace each of them individually. Using a hierarchical process, we combine groups of four sub-frusta together as long as they hit the same primitive.*

traversal cannot stop until the first hit between the scene geometry and the frustum has been computed. Interactive ray tracing algorithms using BVHs also track which rays in the packet are still currently active (i.e. hit the current node) at any point during traversal [145, 93]. Since we want to avoid performing intersection tests with the frustum's sample rays as long as possible, we also keep track of the farthest intersection depth found so far to rule out intersecting nodes that cannot possibly contribute.

Efficient primitive intersection: We currently support triangles as primitives, which is in line with most interactive ray tracers and sound simulations. The main goal for intersection with triangles is to minimize the number of ray-triangle intersections, as they can be more expensive than the traversal steps. Most importantly we want to avoid performing any ray intersections at all if we can determine that the entire frustum hits the primitive, which can happen many times. Consider Fig. 6.4, which shows the different configurations that can arise when intersecting a frustum with a primitive. Case 1 shows that the frustum fully misses the primitives (i.e. no overlap at all); therefore, we can skip that intersection right away. Case 2 shows that the frustum fully hits the primitives, which means we can construct secondary frusta right away without having to consider subdividing the frustum, unless a closer hit is found later on. In cases 3 and 4, the frustum partially overlaps the primitive or contains the primitive and we have to

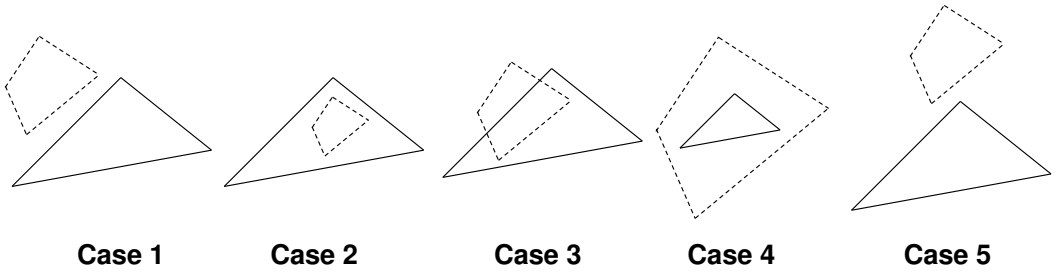


Figure 6.4: **Primitive intersection:** *Five different cases can occur when intersecting a frustum with a triangle. From left to right: Frustum misses completely, frustum is contained, frustum intersects partially, frustum contains triangle. The last case shows a situation where the frustum is clearly outside the triangle, but is not detected by the edge based test since it is not fully on one side of any edge. This case is handled as intersecting, but is culled later on during the clipping test.*

consider the individual sub-frusta.

We test for these four cases by using a Plücker coordinate representation for the triangle edges and frustum rays [133], which gives us a way to test the orientation of any ray relative to an edge. Given a consistent orientation of edges (clockwise or counter-clockwise), we can test for intersection if all the edge orientations have the same sign. When testing the corner rays of the frustum, which can be performed in parallel using SIMD instructions, we check for Case 1 and Case 2 simply by testing whether all the corner rays are inside the triangle (Case 2) or fully outside one or more edges (Case 1). Note that the latter test is conservative and may conclude that the frusta are intersecting the triangle, even if they are not. These intersections will eventually be culled in our handling of Cases 3 and 4. Note that due to the nature of the edge tests, a fifth case is possible (see Fig. 6.4) where the rays are clearly outside the triangle, but not fully on one side of any edge. This case is handled by full intersection.

If no early culling is possible, we then perform a ray-triangle intersection using the actual sample rays. As the number of rays that actually intersect the triangle may be small compared to the number of sample rays representing all the sub-frusta, we first compute the subset of potential intersections efficiently. Since the sample rays are uniformly distributed in the frustum space, we compute bounds on the projected

triangle in that space and only test those samples that fall within those bounds. In order to perform these computations, we clip the triangle to the bounds of the frustum by projecting the triangle to one of the coordinate planes and use a line clipping algorithm against the frustum's intersection with the plane. Finally, when looking at the clipped polygon's vertices, we can compute their bounding box in frustum parameter space (see Fig 6.5).

The actual triangle intersection is only performed for the sample rays that fall within the boundary of the clipped triangle, and can easily be performed by using the indices. Note that this can also be reduced to a rasterization problem: given a triangle that is projected into the far plane of the frustum, we want to find the sub-frusta it covers. Therefore, we can use other ways to evaluate this intersection. By using a higher set-up cost, the triangle could be projected and processed with a scan-line rendering algorithm, intersecting with the respective sample ray for each covered sub-frustum. Another interesting approach would be to use a modified A-buffer [16] for computing the sub-frusta covered by the triangle through lookup masks, at the cost of some precision.

Handling non-specular interactions: As described above, specular reflections and transmissions can be handled directly. Although we have not implemented this, our frustum tracing approach could also use the diffraction formulation described by Funkhouser et al. [42] based on the uniform theory of diffraction. For diffuse scattering the frustum tracing approach could be adapted to also generate secondary frusta on a hemisphere around the hit point. However, this would increase the branching factor per interaction dramatically and generally increase the size of the frusta and therefore have a high negative impact on performance. Since we focus on interactive applications, this currently makes simulating diffuse propagation unappealing.

6.1.4 Sampling and Aliasing

Our algorithm uses a discrete approximation of the exact secondary beams that would be computed by using a full clipping algorithm such as in beam tracing. As a result the reflections obtained by our method can suffer from aliasing artifacts, especially along object boundaries. As shown in Fig. 6.2, reflected frusta often subtend areas that are outside of the primitive or do not cover all of the area. This is due to the fact that our tracing algorithm assumes that a sub-frustum hits the primitive in its full projected area if its sample ray hits the primitive. This can result in other possible effects such as missing paths, e.g. a small hole in the object might be missed due to our sampling density. Fortunately, these artifacts only result in some missed contribution paths from the reflections. Moreover, in a dynamic environment these effects would be far less obvious to the listener as compared to the noise artifacts that can arise due to stochastic sampling in ray tracing methods. Note that our algorithm will also avoid creating holes or overlaps in the reflections field during the computation of reflected or transmitted frusta. These holes or overlaps can have a far larger contribution of error since they tend to be more apparent in an interactive application because of abrupt changes in the contribution. An interesting aspect of our approach is that having small geometric objects or primitives (i.e. a statue) in the scene will not result in a very high number of small secondary frusta. Instead, the number of reflections is bounded by the sampling density in the packet. These very small frusta would be computed by an exact clipping algorithm, though they have very little or no contribution.

One of the main challenges is to compute an appropriate sampling rate (i.e. the number of rays in the frustum). Ideally, the sampling rate could be chosen by taking the highest detail in the scene and setting the frequency so that detail could be reconstructed. Similar to rasterization algorithms, performing this computation in a view-independent manner is almost infeasible due to its high complexity and can lead to very conservative bounds. As a result we use realistic sampling rates and allow some error. There are

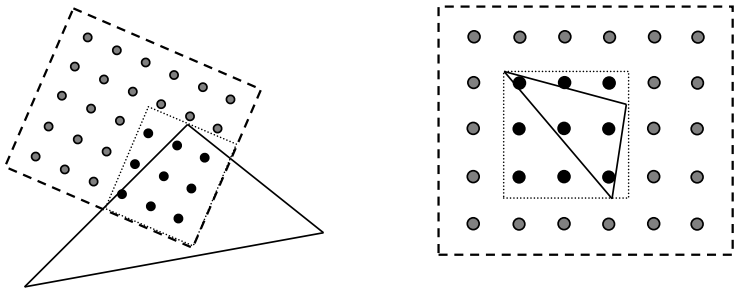


Figure 6.5: **Packet-triangle intersection:** *The intersection algorithm first computes the potential ray intersections in frustum space by clipping the triangle to the frustum's edges in 2-D, then finding the rectangular bounds of the clipped point in frustum space. The bounds can then be used to effectively limit the number of actual sample rays that have to be tested.*

several approaches for choosing the sampling rate in this context: first, a good way of choosing the subdivision is to select the number of rays depending on the angular spread of the packet. For example, a very narrow frustum will likely need a lower sampling density than a wide frustum. Since the actual rays are not constructed until a sufficiently small primitive is encountered, it is also possible to select the sampling rate relative to the local geometric complexity in order to avoid under-sampling. One way to measure local complexity, for instance, would be to use the current depth of the subtree in the BVH. Finally, the sampling rate can also be made dependent on the energy carried by a frustum or the number of reflections before reaching the current position. This is a useful approximation as the actual contribution will likely decrease, and we can lower thesampling rate after a few reflections.

6.1.5 Simulation overview

We now describe an overall sound rendering system that uses our sound propagation algorithm. Our system is designed to be fully real-time and dynamic. We allow movement of the listener, the sound sources and the geometric primitives in the scene by constantly re-running the simulation at interactive rates. The sound propagation algorithm is run as an asynchronous thread from the rest of the system to decouple the simulation update

rate from rendering.

The sound propagation simulation starts out from each point sound source and constructs frusta from that origin that span the whole sphere of directions around it according to a predefined subdivision factor. Each of the frusta is traced through the scene, and secondary frusta are constructed based on the algorithm described in Section 3. There is a user-specified maximum reflection order that limits the number of total frusta that need to be computed. Attenuation and other wavelength-dependent effects are applied according to the material properties per frequency band. Since we regenerate the sound contributions at each frame, we do not save the full beam tree of the simulation, but just the those that actually contain the listener.

Handling dynamic scenes: The choice of a BVH as an acceleration structure allows us to update the hierarchy efficiently in linear time if the scene geometry is animated, or rebuild it if a heuristic determines that culling efficiency of the hierarchy is low (as described in chapter 4). As the BVH is a general structure, our algorithm can handle any kind of scene including unstructured 'polygon soup' and models with low occlusion. Furthermore, we can use lazy techniques to rebuild the nodes of a hierarchy in a top-down manner.

Auralization: So far we have not described how the actual sound output is generated from the simulation algorithm described in the previous section, i.e. the auralization process (we refer the reader to a more detailed overview such as [44] for an introduction). As mentioned above, the simulation is performed asynchronously to the rendering and auralization, i.e. we have a dedicated rendering thread and one or more simulation threads. During the simulation, we do not store the actual frusta, but test each frustum on whether the listener's position is contained in it. If so, we store the sound information such as source, delay and power for all bands in a temporary buffer. The rendering thread reloads this buffer at regular intervals and computes the contribution of each source as an impulse response function (IRF) for each band and

channel. Conceptually, each contributing frustum represents a virtual source located at the apex of the frusta such as in image source methods. Note that this approach can therefore update the sound more often even if the simulation itself is only updated infrequently, which reduces the impact of listener movement.

Furthermore, to incorporate frequency dependent effects, each source's sound signal is decomposed into 10 frequency bands at $20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240$ and 20480 Hz and processed for two channels. For each channel the band-passed signal is convolved with the impulse response for that band and the channel. The convolved signals are then added up and played at the corresponding channel. We also have provision for binaural hearing and we use Head Related Transfer Functions (HRTFs) from a public-domain HRTF database [3]. The sound pipeline is set up using the FMOD Ex sound API. We currently perform all convolutions in software in the rendering thread, but it would be possible to do this in dedicated sound hardware using DSPs as well.

Implementation details: Our ray packet tracing implementation utilizes current CPUs' SIMD instructions that allow small-scale vector operations on 4 operands in parallel. In the context of packet tracing, this allows us to perform intersections of multiple rays against a node of the hierarchy or against a geometric primitive in parallel. In our case this is especially efficient for all intersection tests involving the corner rays as we use exactly four rays to represent a frustum. Therefore most operations involving the frustum are implemented in that manner. The frustum-box culling test used during hierarchy traversal is also implemented very efficiently using SIMD instructions [124]. Finally, since all the frusta can be traced in parallel, performing the simulation using multiple threads on a multi-core processor is rather simple and can be easily scaled to multi-processor machines.

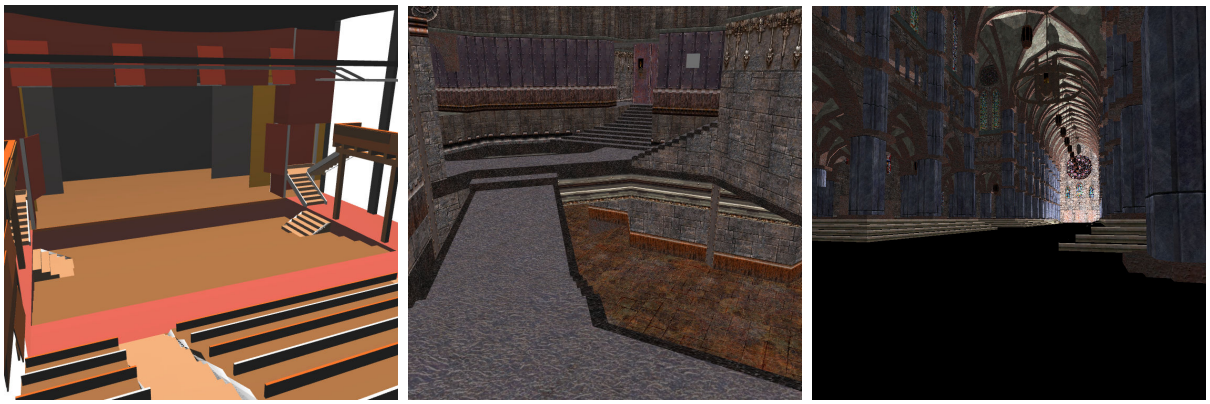


Figure 6.6: **Benchmark scenarios**: *We achieve interactive sound propagation performance on several benchmark models ranging from 9k to 235k triangles while simulating up to 7 reflections. From left to right: Theater (9k), Quake (12k), Cathedral (196k).*

Model	Triangles	Simulation results			Simulation time		
		# sources	# reflections	# frusta	1 thread	3 threads	frusta/thread/s
Theater	9094	1	6	132k	754 ms	276 ms	175k
Quake	11821	3	5	157k	861 ms	290 ms	182k
Cathedral	196344	1	5	60k	1607 ms	550 ms	37k

Table 6.1: **Results:** *This table highlights the performance of our system on different benchmarks. We report timings both for a single thread and three threads as well as in frusta per thread per second. Note that the frustum tracing performance does scale logarithmically with scene complexity and linearly with the number of threads.*

6.1.6 Results

We now present results of using frustum tracing in our system on several scenes. All benchmarks were run on an Intel Core 2 Duo system at 3.0 GHz with a total of 4 cores. As future CPUs will offer more cores, the performance of our sound propagation algorithm can therefore improve accordingly. Results are shown both for using just one thread and using all three threads.

We tested our system on several different environments and conditions (see Fig. 6.6). Our main performance is summarized in table 6.1 and shows that we can handle all of the benchmark models at interactive rates on our test system. The theater model is an architectural scene that is very open and therefore would be very challenging for beam tracing approaches. Even with 7 number of reflections per frustum, we can perform

Model	Triangles	Construction	Update
Theater	9094	319 ms	2 ms
Quake	11821	53 ms	1 ms
Cathedral	196344	1615 ms	26 ms

Table 6.2: **Construction and maintenance cost:** *Our results show that for all the models maintaining or updating the BVH hierarchy adds a negligible cost to the overall simulation. Note that construction only needs to be performed once and then the hierarchy is maintained through updates.*

our simulation in less than one second with dynamic geometric primitives and sound sources. The Quake model was chosen as a typical example of a game-like environment and features densely-occluded portions as well as open parts. Some dynamic geometric objects and moving sound sources are also included in our benchmark. We also tested a more complex, static scene with 190K triangles with just one moving sound source.

The results in table 6.1 show that even though performance as measured by frusta per second decreases with increasing number of primitives, the decrease is still sub-linear. This is due to the logarithmic scaling of ray packet tracing methods. We recompute the BVH whenever the geometric objects in the scene move. Even though the time complexity of updating a BVH is linear in the number of primitives, the total time needed for updating a BVH is still negligible compared to the simulation time, as shown in table 6.2. Moreover, the BVH update can easily be parallelized using multiple threads between the simulation runs.

A key measure in our algorithm is the number of sample rays that are used per frustum. It can have a significant impact on the performance. Figure 6.7 shows the overall simulation performance as well as the total number of frusta used in our benchmark models when changing the sampling rate. The graph shows that the scaling is logarithmic, which is due to the ray-independent frustum traversal as well as our merging algorithm for constructing secondary frusta. This scaling makes the sampling rate a good parameter for trading off quality and runtime performance, depending on the requirements on the simulation.

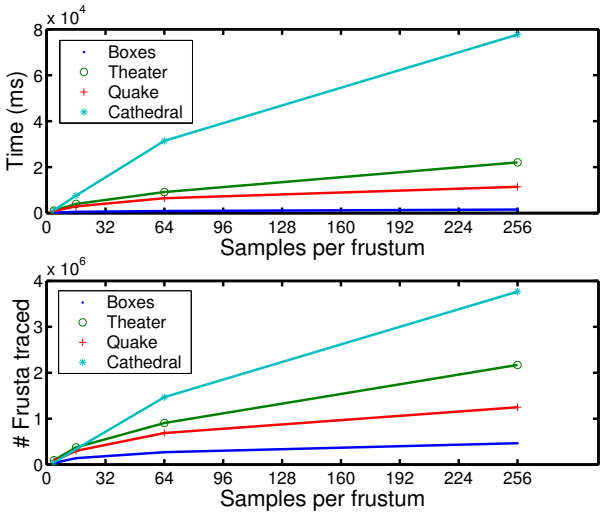


Figure 6.7: **Sampling rates:** *The graphs show the impact of increasing the sampling rate per frustum on both the simulation times as well as number of frusta generated (all simulations are performed for 7 reflections.) In addition to the benchmark scenes used in Table 1, the 'Boxes' scene is a simple environment of two boxes connected by a small opening. Due to our frustum traversal algorithm, efficient triangle intersection and secondary frustum construction, increasing the sampling rate only causes logarithmic growth in the simulation time and number of frusta generated. This suggests that changing the frustum sampling rate can be an efficient method to control the accuracy of our simulation.*

6.1.7 Analysis

We now analyze the performance of our algorithm and discuss some of its limitations. As discussed in section 3 our approach introduces errors due to discrete clipping as compared to beam tracing. We have found that the artifacts created through aliasing are usually hardly noticeable except in contrived situations, and they are far less obtrusive than temporal aliasing that arises in ray tracing algorithms based on stochastic approaches. Note that the sample location in the sub-frusta does not need to be the center, so the aliasing due to sub-sampling could be ameliorated by stochastic sampling of the locations, e.g. by jittering. However, this may introduce temporal aliasing in animated scenes as stochastic sampling may change simulation results noticeably over time. It is possible that Quasi-Monte Carlo sampling could eliminate these problems.

Another source of potential errors stems from the construction of secondary frusta: since the reflected or transmitted frustum is constructed from the corner rays of the sub-frustum, the base surface of the new frustum can significantly exceed the area of the primitive if the incoming frustum comes from a grazing angle and the sample rays hits close to the boundary of the object.

Another limitation of the frustum-based approach is that we assume surfaces are locally flat, and our algorithm may not be able to handle non-planar geometry correctly. This is common to most volumetric approaches, but we can still approximate the reflections by increasing the number of sample rays and using the planar approximation defined by the local surface normal. Our implementation is also currently limited to point sound sources. However, we can potentially simulate area and volumetric sources if the source can be approximated by planar surfaces. The lack of non-specular reflections is another limitation of our approach. For example, it could be hard to create a frusta for diffuse reflection from a surface based on a scattering coefficient without significantly affecting the performance of our algorithm.

We also studied the behavior of our algorithm for different sampling rates. As Fig.

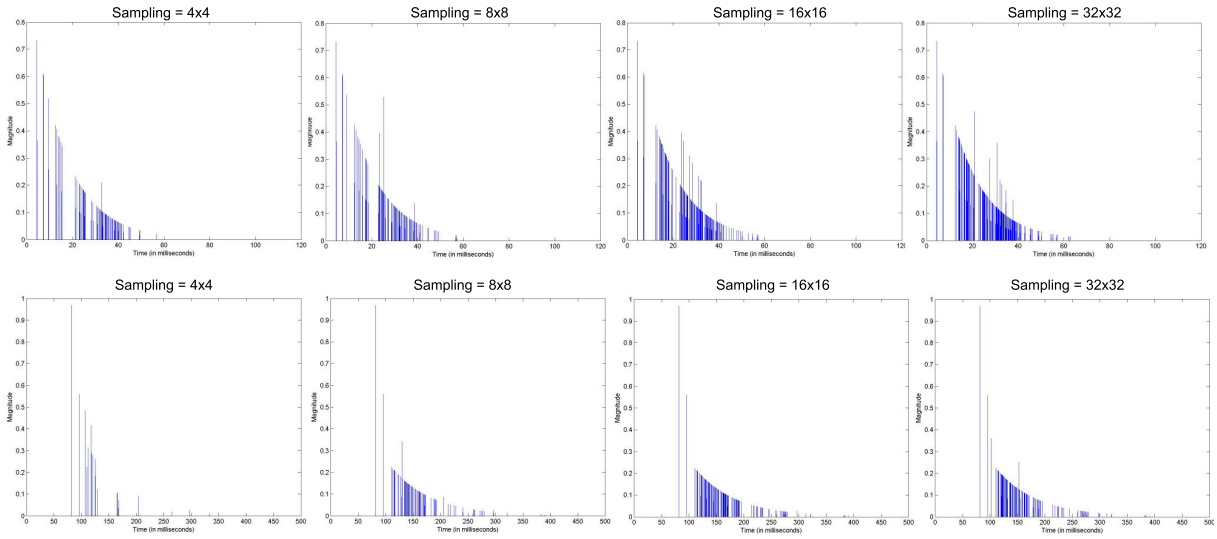


Figure 6.8: **Impulse Response (IR) vs Sampling Resolution**: The above picture shows IRs generated from our frustum-tracing approach for a simple scene of two connected boxes (top) and the Theater scene (bottom), with reflection order = 4 and varying frustum sampling resolution $\{4 \times 4, 8 \times 8, 16 \times 16, 32 \times 32\}$. Notice that the sampling resolution of 4x4 misses some contributions compared to higher ones, but captures most of the detail correctly. As the sampling resolution increases, the accuracy of our method approaches that of the beam tracing method. These results indicate that the accuracy of our method for 4×4 or 8×8 sampling resolution can be close to that of beam tracing.

6.7 shows, the simulation time increases sub-linearly to the sampling rate due to our optimized intersection and sample combination algorithm. Fig 6.8 compares the resulting impulse response functions on two different models for varying sampling rates, which is significant since it is obvious that – as the sampling rate goes to infinity – our algorithm essentially becomes beam tracing. As the results show, even for low sampling resolutions, the response converges very quickly, which suggests that we can achieve almost the same quality with very low sampling rates. Of course, our approach is still a geometric algorithm and like all others its accuracy for high-quality simulation is may therefore be limited compared to full numerical simulation [157].

Note that our frustum tracing technique is could be seen to be related to adaptive super-sampling techniques in computer graphics such as [158, 48, 76]. However, recent work in interactive ray tracing (for visual rendering) has shown, that adaptive sampling – despite its natural advantages – does not perform near as fast as simpler approaches that are based on ray packets and frustum techniques. While high uniform sampling, as used in our algorithm, may seem uneconomical at first, our clipping algorithm reduces the actual work and simplicity makes this approach map much better to current hardware. Combining samples only after the sampling has been performed reduces the detail of the uniform sampling to the same that adaptive sampling would generate, but does not add any overhead to the traversal process. Similarly, there were parallel approaches in other areas such as radio [120] and sound propagation [139, 32] using adaptive beam methods, but for the same reasons they do not perform nearly as well and are limited in the scale and generality of scenes they can handle.

6.1.8 Conclusion

In conclusion, the frustum tracing approach shows that ray tracing techniques for visualization can be adapted and used for other geometric simulation methods as well. In particular, BVHs offer many advantages including a robust traversal for general pack-

ets and frusta as well as support for dynamic environments as presented in previous chapters. Even though the frustum tracing algorithm introduces several algorithmic improvements, the core of the algorithm still is very similar to that of a normal ray tracer in rendering, and it is very likely that further improvements in interactive ray tracing will improve the state-of-the-art in sound simulation as well.

Beyond ray tracing, BVHs and object hierarchies in general have other very different applications, one of which will be described in the next section.

6.2 Parallel collision detection on GPUs

Collision detection is widely used in computer graphics, physically-based modeling, virtual reality, haptics and robotics and is important to perform accurate simulations. Some of the most common algorithms use bounding volume hierarchies to accelerate different queries such as collision of models or self-collision operations that test for intersection of all triangles in a model.

In chapter 5 we demonstrated parallel algorithms for constructing and refitting AABB bounding volume hierarchies on GPU architectures. In this chapter, we show how these algorithms can be used to construct more complex bounding volumes such as oriented bounding boxes (OBBs) with very little overhead. In addition, we present a better approach to distributing work over GPU cores than the work queue compaction approach used in that chapter. We demonstrate that this also enables the implementation of parallel collision detection queries that intersect hierarchies.

6.2.1 Background

Bounding volumes for collision

BVHs have been widely used for accelerating collision and distance queries. These include discrete as well as continuous collision detection, including self-collisions. The

hierarchies mainly differ in the choice of the bounding volume (BV). The simplest hierarchies use simple BVs such as spheres or AABBs, which have a lower storage overhead, lower cost for updating the hierarchy and performing overlap tests. Other BVHs use tight fitting BVs such as K-DOPs, OBBs and RSS [35]. These BVHs have a higher storage overhead and increased cost of overlap test. However, their culling efficiency is much higher than the BVHs based on simple BVs. In case of rigid models, the BVHs are computed once and are traversed at run-time to resolve the query. In this case, some of the fastest collision algorithms use tight-fitting BVs such as K-DOPs [84] and OBBs [51] and the fastest separation distance computation algorithms are based on RSS [92]. However, for deformable models, the cost of reconstructing or updating the hierarchy at each step of the simulation can be high. Therefore, most CPU-based algorithms for deformable models use AABB or sphere hierarchies [144, 14, 24]. However, these simple BVs can result in a high number of false positives and the resulting algorithms perform a high number of elementary tests [24].

Fast hierarchical collision detection

Hierarchical techniques based on BVHs have long been a popular choice to accelerate collision queries. Most recent improvements to these queries have either come from improved culling techniques [24, 141] or parallelism [80]. However, there is a widespread notion that it is hard to implement these hierarchical algorithms on GPUs. Instead, one approach has been to use the GPU for broad-phase collision only [95]. Alternatively, many GPU-based collision checking algorithms exploit the rasterization capabilities of GPUs by using depth or stencil buffer tests at image-space resolution [64, 85, 55]. In order to handle complex models, the hierarchical traversal is performed on the CPUs [53, 140] and this results in additional CPU-GPU data transfer overhead. A similar approach combines both multi-core CPUs and GPUs [81].

There is considerable literature in parallel computing on the use of work queues for

load balancing, including locking and non-locking shared queues such as work stealing approaches [7, 65]. These techniques map very well to hierarchical and recursive operations and have been employed extensively in parallel systems and parallel programming languages such as Cilk [41]. However, they have not been used on GPUs as the overhead of performing communication between the cores through main memory can be rather high. Instead, previous techniques for GPU work queues used explicit compaction methods between kernel calls. The overhead of these methods makes them efficient only for applications with relatively high computational intensity and coarse-grained parallelism [166, 94]. There are known parallel algorithms to accelerate hierarchical traversals [121, 87] and they are also applied to parallel collision detection [83, 56]. However, most of these methods either perform communication between the processors or are not suited for GPU-like architectures.

Challenges

In case of hierarchical collision detection and distance queries, the major challenge is that work is generated dynamically as the algorithm progresses and that the computational load can change significantly. Thus, any parallel hierarchical algorithm needs to address the problem of load balancing and work distribution in order to maintain availability of parallelism for all cores. On multi-threaded CPU architectures, prior approaches have used work queues and work stealing for operations with hierarchies and recursion with similar properties [7]. However, these techniques do not currently work well on GPUs for multiple reasons. Primarily, they are based on the assumption that low-latency communication between cores is possible in order to manage concurrent access to shared data structures. Unfortunately, this is only possible in a very restricted sense on current GPUs. The main barrier to communication is the latency and lack of a memory consistency model in the global GPU memory shared by the cores, i.e. different cores are not guaranteed to see memory writes from other cores at the same time or

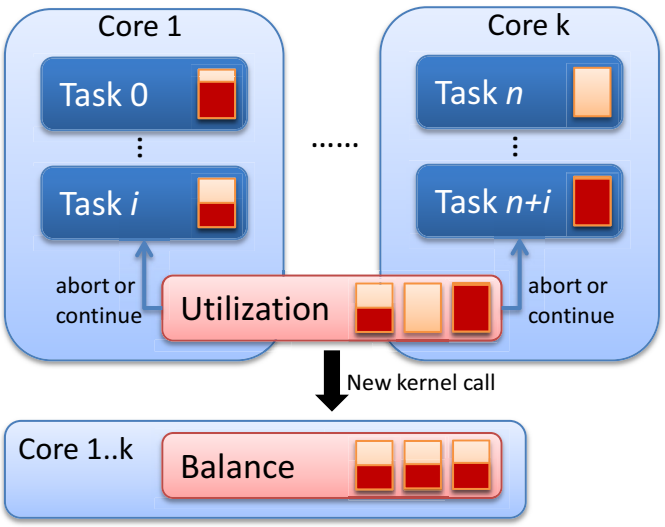


Figure 6.9: **Load balancing for hierarchy computation and traversal:** In our approach, each task keeps its own local work queue in local memory and can generate new work units (such as intersection operation) without coordinating with others. After processing a work unit, each task is either able to run further or has an empty or completely full work queue and wants to abort.

may not even see them in the same order they were written. Even though newer GPU architectures provide atomic operations such as compare-and-swap (CAS) that could be used for locking operations, the remaining problem is that previous writes to the memory protected by the lock may not have been executed yet, thus preventing implementation of work queues or other structures shared by all cores. Even if memory consistency was not a problem, busy waiting such as by spinning on a lock variable is relatively inefficient on an architecture with high memory latency and hardware multi-threaded execution can also lead to priority inversion and prevent other threads on the same core from performing useful work. As a result, one of the major challenges in terms of hierarchical traversal is to balance the load evenly among multiple cores on the GPUs without large synchronization overhead.

6.2.2 Lightweight work balancing

In the context of this chapter, hierarchy operations include work such as testing a pair of nodes for intersection or computing a separation distance. As a result of that test, new pairs of nodes may have to be tested afterwards. We refer to the information describing the specific test to be done (e.g. references to two nodes) a *work unit* in the further discussion. In order to efficiently parallelize the hierarchy operations, we describe a novel approach that distributes these work units between GPU cores and threads efficiently. The main goal of our technique is to minimize the amount of synchronization overhead while performing actual work. In our approach, we launch a number of parallel tasks that run on separate cores. Every task keeps its own work queue either in the local memory or global memory, depending on its size constraints. These queues are only accessible locally; thus, no synchronization is necessary between cores when reading or writing to queues. Work kernels can remove an element from the queue and then create new ones as well. In order to use the vector processors, we also provide implementations for data-parallel access to the queues such that the kernel may work on multiple work units in parallel For example, on a 32-wide vector unit, at each step up to 32 work elements are dequeued, processed and then some number of new units is pushed back onto the queue. We can use either local atomic memory operations or explicit reduction steps to synchronize access to the local work queue.

The main step that ensures that all cores have work is the balancing step (also see Fig. 6.9). Synchronization is only performed on one global variable that counts the number of cores that cannot do further work (i.e. the queue is empty or full). Whenever a task reaches that state, it atomically increases the variable and terminates the kernel. After executing a work unit, each task tests the current value of the variable and compares it against a user-specified idle threshold. If higher, then it terminates and writes back its local queue to the global memory; otherwise, it continues. After all cores have either finished or aborted, we run a kernel that examines the work queues from all the cores,

then assigns a roughly equal number of work units to each. If there are work units to process left, then the work kernel for the current hierarchy operation is called again and the process repeats. Note that the number of actual tasks is in fact larger than the total number of GPU cores, to allow for hardware multi-threading. Therefore, even if some percentage of tasks have been aborted, the core it was scheduled on may not be idle but just processing another task. In practice, we have found a threshold of 50% idle tasks for balancing to work best.

6.2.3 Hierarchy traversal for collision detection

We use BVHs to check for collisions between two disjoint objects (inter-object collisions) as well as self-collisions for deformable objects (intra-object collisions). We assume that each object is composed of triangles and we do not make any assumptions about their connectivity. As a special case, self-intersection involves checking whether any of the non-adjacent triangles of the object intersect each other, as is needed in cloth simulation or surgical simulation. Since many triangles can be thin with large aspect ratios, overlaps can be missed if discrete collision checking is used. Therefore, *continuous* collision detection (CCD) algorithms are used to check whether there is a collision between the discrete time instances, and is more expensive than discrete collision checking. In practice, the hierarchy is built on top of each object's triangle or the swept volume between the time instancesin CCD. During each step of the simulation, we use the hierarchies to compute the potential collisions and only perform triangle-triangle overlap tests on those candidate pairs.

Simultaneous hierarchy traversal

The traversal algorithm starts with the two BVH roots and tests the BV for overlap. If the BVs overlap, then all possible pairings of their children are recursively tested for intersection. If both of the nodes are leafs, then the two corresponding triangles are

put on a list of potential intersections. If only one of the two is a leaf, then it is tested against the children of the other node. This can be seen as traversing a tree of possible bounding volume node pairs, also called the bounding volume test tree [92] (BVTT.) Implicitly, our algorithm is performing a parallel traversal of the BVTT.

The main work units are pairs of hierarchy nodes and for a binary tree each intersection test can generate up to four pairs per step. All intersection tests between the nodes of the hierarchy can be performed independently. The intersection kernel can be run using the vector units to process several intersections in parallel and push the resulting new intersection pairs on the work queue or in a separate result queue for actual triangle pairs. After this traversal, the overall list of triangle pairs is then used as input into an intersection test kernel that tests for actual overlap. Because all potential intersections can be tested fully in parallel, this step is simple to implement by starting enough tasks for all the pairs.

BVTT traversal

One problem with this approach is that there is a lack of available parallelism while testing the higher levels of the hierarchy that can reduce the overall performance of the algorithm. However, it is possible to exploit temporal coherence in the traversal and drastically increase the level of parallelism. In many interactive applications there is considerable temporal or spatial coherence between successive time steps; this can be exploited by front tracking [33, 84]. We keep track of the front in the BVTT (see Fig. 6.10) which consists of all the intersecting leaf node pairs of the BVTT as well as every non-intersecting node pair for which a sibling overlaps. This simple list of node pairs can be generated during the BVH traversal. For the next frame, we use this list as input for the work balancing kernel and use the same pairs as the starting work units. The traversal kernel for processing this front is a modified version of the standard traversal algorithm: for each initial node, the intersection test is performed again with

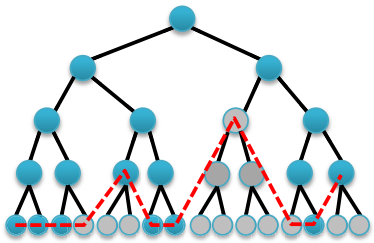


Figure 6.10: **BVTT front:** During the intersection operations, only a subset of the BVTT nodes is explored (in blue), while others (grey) are not explored as their ancestors do not intersect. By tracking the front of blue nodes in the (implicit) hierarchy from the last collision query, we can use it as a starting point for the next query.

the updated BVs. If the pair had an overlap during the last frame and intersects again, then the triangle pair is written to the intersection queue. If the pair did not intersect last frame, but does now, then new work units are created as in the normal traversal. Finally, if the pair does not intersect, then the kernel loads the parents of both the nodes and tests them for intersection. If they still intersect, then we have at least one sibling that must still overlap and the node pair is kept in the front. Otherwise, the front is moved upwards by adding the pair of parent nodes to the work queue for the next step. However, since all the siblings are in the front by definition, the parent pair would be added to the list multiple times, which is to be avoided. Instead, we check whether the pair is the leftmost child in the BVTT and only add the parent pair if that is the case to avoid duplicates in the list.

6.2.4 Computing bounding volumes for collision detection

Tight-fitting bounding volumes have been shown to provide much higher culling efficiency during distance and collision queries [51, 84, 92]. However, most recent CPU-based algorithms for deformable models tend to use simple BVs such as AABBs because of compact storage and lower cost of refitting or hierarchy computation. Interestingly, this trade-off changes on GPU architectures as they reward the much higher ratio of computational power to memory latency. However, in order to use OBBs it is necessary

to also compute the OBB hierarchy efficiently. We solve this problem through parallel hierarchy refitting.

Parallel hierarchy refitting

The hierarchy refitting approach that was discussed in chapter 4 assumes that a hierarchy already exists, but that primitives may have been moved or deformed, and it will modify the bounding volume associated with each node of the hierarchy to reflect the newly changed primitives. In this case, the primitives associated with each node of the tree do not change and the work structure is simple since it simply needs to perform a post-order traversal of the hierarchy. At each step, the main work is updating the bounding volume of each node based on the bounding volumesof the children. At the leaf nodes, the bounding volumes are computed from the actual geometric primitives. In a normal recursive implementation, the traversal thus goes down the tree to the leaves, then propagates the new bounding volumes upwards to be merged until the root is reached.

It would be possible to express this in our framework by issuing two different tasks, a downward traversal and an upward merge task. However, since the actual computational load is already very small, we found that it is faster to reorder the layout of the tree in memory such that the nodes are stored by tree level. For our parallel BVH construction approach, this is very simple since the nodes will already be stored in that order due to the breadth-first construction order. All that remains is to store a start offset and node count per level to prepare for refitting. We start out at the leaves of the tree and only need to perform the merge kernel. Since each level is dependent on the results of the previous one, it is necessary to terminate the computation after each level and then call the merge task again with the front set to one level further up. In other words, no coordination besides a barrier between tree levels is needed and no explicit management of work units is necessary since the structure is known in advance.

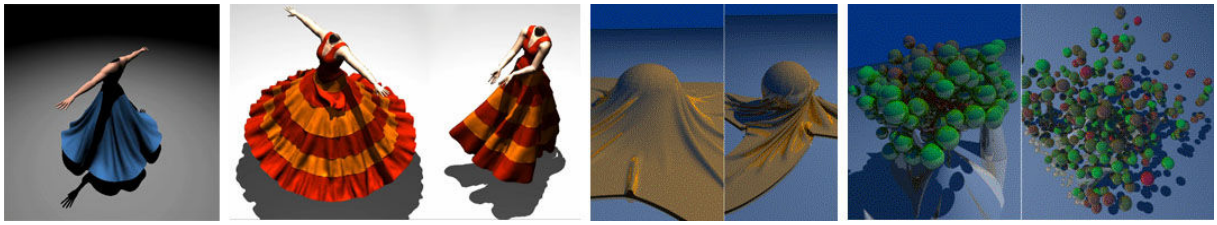


Figure 6.11: **Benchmarks:** The benchmark models used for collision detection in this order: Princess cloth simulation (40K triangles); Flamenco cloth simulation (49K triangles), Cloth dropping on sphere (92K) and n-body simulation (146K). Our algorithm can perform interactive continuous self-collisions using OBB hierarchies and pairwise elementary tests on all of these models in tens of milliseconds, about $7-12$ faster than prior methods.

Computing OBB trees

The refitting approach allows us to reduce the amount of hierarchy maintenance for deformations between collision detection steps since we can avoid having to fully rebuild the hierarchy which might dominate the hierarchy intersection time. However, it also provides a tool to quickly compute a hierarchy with OBB or other bounding volumes: first, we build a standard AABB hierarchy using the algorithms described in chapter 5. Then, we run the refitting algorithm on the hierarchy, but instead of recomputing the AABBs we compute the OBBs for the same tree structure. In our results (see the next section) this typically only adds about a 20-25% overhead to the overall construction cost.

In further results, we use OBBs (and compare against AABBs) in our implementation. We use the fitting method based on principal component analysis as well as the separating axis overlap test [51] for triangles, then merge OBBs in the refitting process. Even though these operations involve using about $1-2$ orders of magnitude more instructions than AABB trees, the overall parallel computation and higher culling efficiency of OBBs results in improved overall performance on GPUs.

Model	Tris	Build		Refit	
Collision		AABB	OBB	AABB	OBB
Flamenco	49k	22	27	1.73	4.6
Princess	40k	20	24	1.4	3.9
Sphere/Cloth	92k	31	39	2.5	7.9
Balls	146k	56	68	3.2	11.5

Figure 6.12: **Parallel hierarchy results:** The benchmark scenes used in this paper, and timings (in ms) for our construction and refitting algorithm both for AABB and OBB hierarchies. Note that the overhead for creating and refitting a hierarchy with more complex bounding volumes is relatively low.

6.2.5 Results

We have implemented our approach using a Intel Core2 Duo system at 2.83 GHz on 4 cores. We use CUDA on a NVIDIA GTX 285 GPU that has a total of 30 processing cores and 1 GB of memory. Our collision detection algorithm uses a variant of the Moeller test [106] for discrete triangle-triangle intersection. For continuous triangles, we solve the cubic equation [118] for each of the 15 elementary vertex/face and edge/edge tests to compute the first time of contact.

We use several commonly benchmark scenes for collision and distance queries and compare their performance with prior methods (see Fig. 6.11). These models range from 12k to 245k triangles each and can have multiple triangle pairs in close proximity. For example, the Flamenco model has several cloth layers very close together, representing a hard case for culling in collision detection algorithms. Figure 6.12 summarizes the results for our parallel GPU construction and refitting algorithms. We also compare the timings for building a AABB BVH to show the overhead of OBBs. Since refitting cost is about an order of magnitude lower than construction, it is a good choice for handling deformable models and animations with no topological changes.

The collision detection performance is summarized in Fig. 6.13. We provide results for discrete as well as continuous versions using either AABB or OBB bounding volumes. In addition, we show the impact of exploiting temporal coherence by using our front-

Model	Discrete		Continuous	
	AABB	OBB	AABB	OBB
Flamenco	35	29	40	38
Princess	22	28	27	34
Sphere/Cloth	33	49	47	43
Balls	85	75	99	81

Model	Discrete		Continuous	
	AABB	OBB	AABB	OBB
Flamenco	27	22	37	34
Princess	17	22	26	29
Sphere/Cloth	28	36	42	38
Balls	78	70	91	74

Figure 6.13: **Performance results:** These tables highlight the performance of our collision detection algorithm (top: without BVTT, bottom: with BVTT), which checks for inter-object and intra-object collisions. All the numbers are in milliseconds and include the time for refitting, front-based traversal and pairwise triangle intersections.

based traversal implementation of the same algorithms. All the numbers are averaged over the whole animation. Note that AABBs are slightly faster on most benchmarks for discrete collision detection where intersection tests are relatively cheap. However, for continuous collisions where better culling performance is more important, OBBs provide better performance despite their higher cost for maintenance and traversal. The front-based traversal generally results in a speedup compared to full traversal, although the result is model-dependent since temporal coherence can vary. The overall impact of any front caching method is limited since by definition at the very minimum all the leaf in the BVTT need to be tested, which will be half the total tested nodes in any case. Thus, even for an unchanged frame the theoretical work reduction would be half the number of BV overlap tests, with no decrease in primitive-primitive tests.

We also look at a breakup of timings within the collision detection algorithm, i.e. refitting, traversing, balancing and triangle intersection (see Fig. 6.14), in the Flamenco model. The results show that a large part of the time is spent in the traversal part while intersection tests account for a smaller percentage, mostly due to the fact that it runs with optimal parallel utilization given that all intersections can be performed independently. Refitting takes a relatively constant fraction of time.

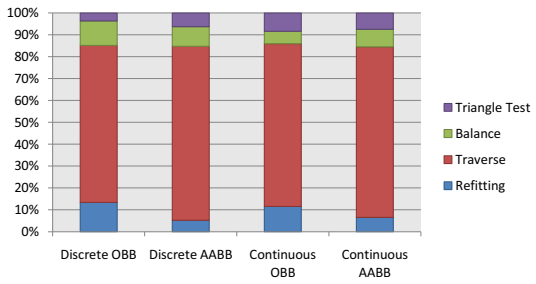


Figure 6.14: **Split-up of timings:** The fraction of time spent in the parts of the algorithm differ based on whether continuous or discrete collision detection is performed and the choice of the BV. In general, the use of OBBs results in more time spent in refitting and traversing, but less in intersection tests due to higher culling efficiency.

6.2.6 Analysis

The performance results show interesting implications for the choice of BVs for collision detection on GPUs. Most current CPU approaches use AABBs since they provide very fast intersection and refit operations and are relatively compact in memory. Our results show that for discrete collision detection AABBs can provide improved performance in many cases. However, for continuous collision detection this situation is reversed and OBBs provide faster results. We have found that OBB intersection and refitting benefits from having much higher *compute density* that is a good match to the high computational power of GPUs. In particular, OBBs use 2.5 as much memory as AABBs, but operations such as computing an OBB from triangles or intersecting two OBBs take about two orders of magnitude more instructions. For example, AABB-AABB intersection needs just 6 comparison operations and needs to load 12 coordinates to do so. In contrast, OBB-OBB intersection test loads 30 coordinates, but then needs to perform 15 separating axis tests, resulting in hundreds of operations. This will be slower on a low-latency CPU architecture with less compute power, but fast on a GPU architecture where high-latency memory accesses are expensive and computational power is high.

Limitations: Our approach has some limitations. Firstly, getting high performance or speedups for collision detection depends on having a relatively large front in the

BVTT traversal tree. Unlike self-collision, inter-object collision between two different hierarchies, e.g. deformable models, may exhibit a much smaller front unless the objects are very close such that there are many BVs overlaps. For very small models, available parallelism may be limited. In this case, handling multiple object queries in parallel will be a better solution to exploit the capabilities of a GPU. In general, even though our approach tries to implement very lightweight synchronization, we are still limited by inherent memory latency of GPUs for communication. To achieve better scaling, our approach could benefit from having a low-latency communication channel between the cores on GPU architectures. This would allow the implementation of algorithms such as work stealing during kernel execution, which based on our experiments can already be implemented on current GPU architectures, but are very slow. Future GPU architectures that have coherent caching could also improve performance significantly.

6.2.7 Comparison

In this section, we compare the performance of our algorithms with prior CPU-based and GPU-based algorithms.

Some of the fastest CPU-based algorithms for continuous self-collision use feature-based hierarchies and other culling methods (e.g. normal cone tests) to reduce the number of pairwise intersection tests. As an example, the representative triangle algorithm [24] takes about 200ms per frame on a single core for continuous self-collision detection on the Flamenco model, as compared to our algorithm that takes about 35ms to perform a query. Note that these culling approaches are orthogonal to our work and could be integrated into our framework as well. Some recent algorithms have implemented the hierarchical CCD test on multi-core CPU systems [80] and performed continuous self-collision on the cloth/sphere benchmark in 53ms (vs. our 34ms) using an 8-core Xeon system. A more recent hybrid version running on 4 CPU cores and 2 GPUs improves timings to only 23ms, but uses more computational resources [80].

Several previous approaches have been proposed to perform self-collision on GPUs using rasterization algorithms [64, 85, 55]. Govindaraju *et al.* [53] used occlusion queries along with CPU-based overlap tests to perform continuous self-collisions for cloth simulation and were able to handle a 13K triangle version of the 40K Princess model at about 500ms. In contrast, our approach tests the same model with three times the complexity thirty times faster, though we use a faster GPU. Similarly, Sud *et al* [140] perform self-collisions by using discrete Voronoi diagrams generated by rasterization. Their approach took 800ms ($\tilde{1}$50ms scaled by FLOPs) on a 15K version of the Cloth/Ball scene we use, whereas our approach is over 25x faster on the full 92K model. Note that it is hard to compare performance across GPU generations. In addition, previous approaches relied heavily on occlusion query performance and CPU-GPU bandwidth which has not scaled with the computational power of many-core GPUs.

6.2.8 Conclusion

In conclusion, parallelizing recursive operations such as hierarchy intersections on GPU architectures is possible and in current implementations far preferable to previous rasterization-based approaches. We have shown that the focus on compute intensity actually leads to different choices compared to multi-core CPU systems. Overall, performance is already very competitive, but similar to hierarchy construction it is unlikely that a high fraction of peak compute performance will be useable without architectural changes for work distribution.

6.3 Selective ray tracing for hybrid shadows

In this final application, we further investigate the possibilities opened by being able to perform interactive construction and maintenance of hierarchies on GPUs, in this case the advantages of also having very fast rasterization capabilities. Many rasterization-

based approaches can be used to generate shadows on GPUs, but when evaluating them on large, complex models they typically only provide either high performance or accuracy, but not both. On the other hand, ray tracing provides a simple solution to generate accurate hard and soft shadows with good scalability for large in-core models. Despite recent advances and good use of parallelism, for rendering current ray tracing implementations are one or two orders of magnitude slower than rasterization approaches on GPUs.

In this chapter, we present a *selective ray tracing* algorithm to generate accurate hard and soft shadows on current many-core GPUs. Our approach is general and the overall image quality is comparable to that of a fully ray-traced shadow generation algorithm. Moreover, unlike previous similar approaches [10, 66] the algorithm scales to handle highly complex models, as long as the model and its bounding volume hierarchy can fit into GPU memory. We also show how the ReduceM data structure introduced in chapter 3 can be used for massive model ray tracing and rasterization on GPUs.

6.3.1 Shadow algorithms on GPUs

We give a brief overview of related work limited to interactive shadow generation and reducing aliasing errors and refer the readers to [59, 98, 90] for recent surveys on shadow algorithms. At a broad level, prior techniques to alleviate aliasing artifacts using rasterization methods are based on shadow maps [160] and shadow volumes [23]. Some hybrid approaches have been proposed that combine shadow mapping and volumes [103, 18] and can improve shadow volume performance and allow interactive high-quality shadows on simple scenes. Most current interactive applications use variants of shadow mapping, but may suffer from aliasing problems. Many practical algorithms have been proposed to alleviate perspective aliasing [136, 161, 98] as well as projective aliasing [96]. Other shadow mapping algorithms can eliminate blocking artifacts [1, 75] by implementing a rasterizer that can process arbitrary samples on the image plane.

In general, soft shadows can be implemented by sample-based methods such as using averaging visibility from multiple shadow maps to calculate visibility or ray tracing. Both these methods can be slow, so many approaches have been developed to generate plausible soft shadows with methods such as post-filtering shadow maps or special camera models [105], which produce correct results only for simple scenes. More accurate approaches evaluate the light source visibility from the image samples by back-projection [8, 129, 134, 9] or by generating shadows from environment lighting [4]. Techniques using irregular z-buffering have also been extended for soft shadows on a proposed new architecture [74]. Several approaches combine rasterization and ray tracing to generate higher quality images [137] or use GPU rasterization for visibility and shading [40, 69]. Two recent methods are most closely related to our approach. Beister *et al.* [10] combined shadow maps and ray tracing, but used CPU ray tracing, whereas Hertel *et al.* [66] also ray traced on the GPU. We compare our algorithm more closely to these methods in section 6.3.5.

6.3.2 Selective ray tracing

In this section we give a brief overview of our hybrid rendering algorithm that performs selective ray tracing. Most prior work on hybrid combinations of rasterization and ray tracing has been used for corrective textures [137] or other rendering applications. We perform *selective ray tracing*, and only shoot rays corresponding to a small subset of the pixels in the final image. This is in contrast to *full ray tracing* that shoots rays through all the pixels. Our assumption is that shadow mapping computes the correct shading information for most pixels in the frame, but may include localized error such as aliasing artifacts in parts of the image. We try to identify these regions of *potentially incorrect pixels* (PIP) in a conservative manner since any additionally selected pixels will not change the image whereas missed ones may result in artifacts. As Fig. 6.15 illustrates, this is a multi-step process that starts with the results of a GPU rasterization

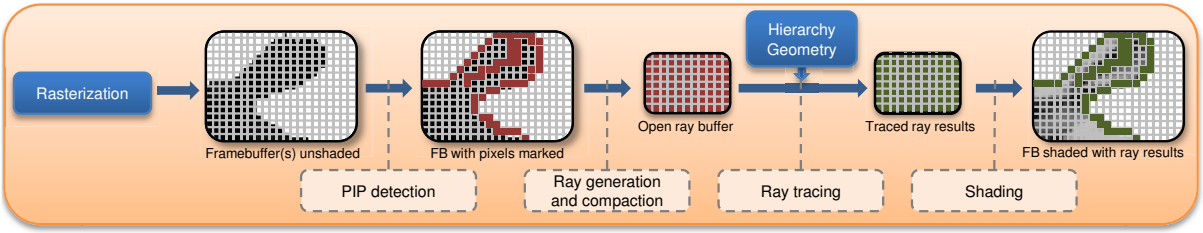


Figure 6.15: **Overview:** *Pipeline model of our hybrid rendering algorithm. After GPU-based rasterization is run, the PIP computation detects and marks pixels that need to be ray traced. The ray generation step generates a dense ray list from the sparse buffer of potentially incorrect pixels and then generates one or more rays per pixels as required. A selective ray tracer traces all the rays using the scene hierarchy and then applies the results to the original buffer. Finally, the pixels are shaded based on the ray results.*

algorithm that provides a first approximation to the desired result (e.g. shadows). As a next step, we test the accuracy of each pixel and classify it accordingly, marking each pixel in a buffer, e.g. as pass/fail. The buffer with all marked and unmarked pixels is then passed into the ray tracer where the first step filters out all non-marked pixels and keeps just the potentially incorrect pixels. For each pixel in the PIP set, we shoot one or more rays to compute the correct visibility or shading information for the underlying problem (e.g. shadows.) All rays are stored in a dense list that can be used as input for any data parallel, many-core GPU ray tracer. After the rays have been evaluated, the results are then written back to the original pixel in the PIP set. Back in the rasterizer, a shading kernel is used to compute colors for all the pixels.

In general, this hybrid approach is one of the ways to complement the weaknesses of current GPU-based rasterization algorithms. The most important issue that governs its performance is the fraction of pixels that are in the PIP set. If only a small number of pixels are correct then the overhead of the hybrid approach may outweigh the savings compared to full ray tracing. This also means that the PIP computation algorithm should try to limit the number of pixels it marks while still being conservative. Also, any rasterization algorithm that is used as input for the first step should be efficient so that it is able to compute visibility and shading information for the initial image quickly

even on complex models. Otherwise, the typically logarithmic scalability of ray tracing may make a full ray tracing solution faster. The next section describe our algorithms for shadow generation that are based on these characteristics.

6.3.3 Shadows using selective ray tracing

In this section, we present our algorithms for applying the selective ray tracing approach generating high quality shadows based on hybrid rendering and selective ray tracing.

Hard shadows

Shadow mapping is one of the most widely used algorithms for generating hard shadows for interactive applications. It works on general, complex 3D static and dynamic scenes and maps well onto current GPUs. However, the shadow maps may need high resolution to avoid aliasing artifacts. These errors can be classified into *perspective* and *projective* aliasing [136]. Perspective error occurs due to the position of the surface with respect to the light and viewpoint and result in the 'blockiness' of shadows in the algorithm. Projective error stems from the orientation of the receiver to light and viewpoint. Geometric errors from under-sampling can also include missing contributions from objects that are too small or thin in light space, e.g. surfaces that are oriented close to coplanar with the view direction. Artifacts from this error result in missing and interrupted shadows for these objects. Finally, shadow map self-shadowing error occurs from inaccuracies in the depth values computed in the light view and the camera view. It stems both from depth buffer precision (numerical error) as well as orientation of the surface (geometric error). We consider this mainly an artifact that can be minimized by increasing depth precision and using slope-dependent bias and do not address it directly in our algorithm.

Pixel classification: We now discuss our method for estimating the set of pixels in the image that can potentially be incorrect (PIP) due to the error described above. We first note that most of the error in shadow mapping appears at shadow boundaries

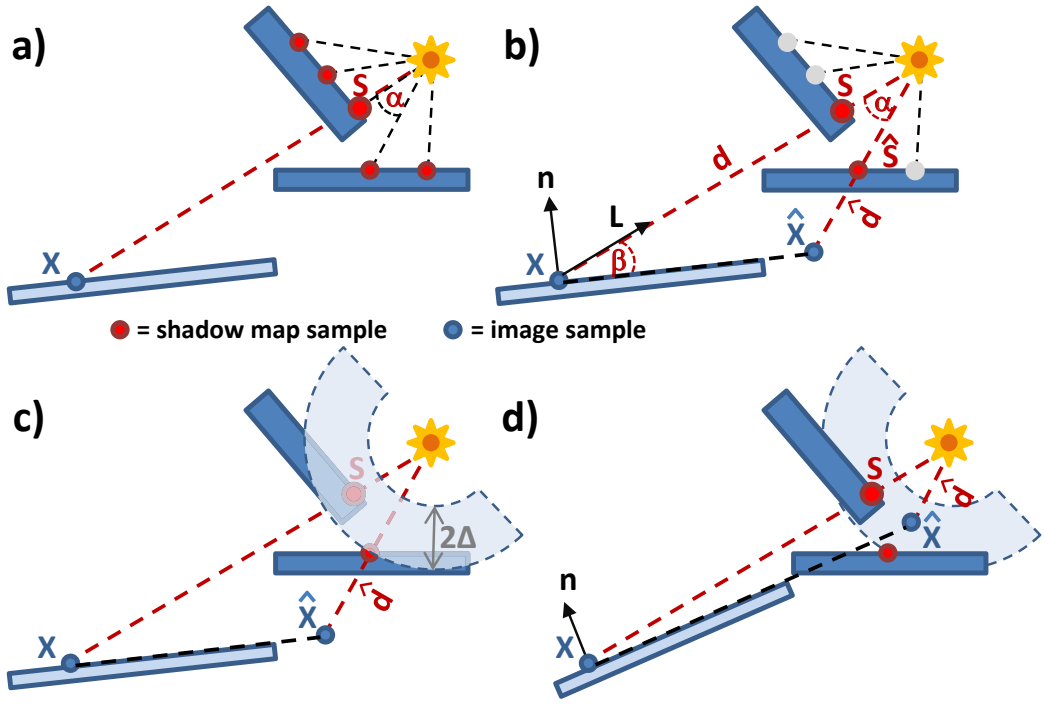


Figure 6.16: **PIP computation:** *a) For a given image sample* $\boldsymbol{X}$ *we project back to the shadow map and find the corresponding sample* $\boldsymbol{S}$*. We then test the depths of the surrounding shadow map samples and select the one with the maximum difference* Δ *in depth value to* $\boldsymbol{S}$ *and label it as* $\hat{\boldsymbol{S}}$*. b) We now determine whether the surface at* $\boldsymbol{X}$ *can be affected by an edge at* $\boldsymbol{S}$ *and* $\hat{\boldsymbol{S}}$ *by finding the closest point* $\hat{\boldsymbol{X}}$ *on the surface within the angle* α *of one shadow map pixel and find its depth* $\hat{d}$*. c) If* $\hat{d}$ *is within the shaded region of depth* Δ *around* $\boldsymbol{S}$ *or on the other side of the region as seen from* $\boldsymbol{X}$*, then does not need to be ray traced. Here, the pixel can be classified as shadowed. d) Counter-example:* $\hat{\boldsymbol{X}}$ *is in the region and thus the pixel is ray traced.*

while the shadow interiors (as well as the interiors of lit regions) tend to be accurate. Thus, we can assume that when we look up the corresponding sample in the shadow map for a given image sample, it should not be considered accurate if it is adjacent to an edge in the shadow map. We therefore modify the standard depth buffer look-up to test the 8 texels around the sample value with depth s in the shadow map and find the maximum absolute difference $\Delta = \max(\|s - s'\|, s' \in \text{depth around S})$ in depth. If Δ exceeds a threshold value, e.g. a fraction of the possible scene depth as determined by z-buffer near and far planes, we assume that there is a shadow edge at this image pixel. In this case, we need to make sure this edge can affect the shadow generation at the current shading sample $\mathbf{X}$ that is at distance d from the light source (see Fig. 6.3.3 for illustration.) This is to prevent that a relatively small shadow discontinuity at one side of the model does still affect pixels far away. To achieve this, we find the closest distance to the light $\hat{d}$ that the receiver surface can reach within the angle α of one texel of the shadow map. Intuitively, the closer to parallel the receiver surface is to the light direction, the larger the difference between $\hat{d}$ and d becomes. If $\hat{d}$ overlaps the interval $[s - \Delta, s + \Delta]$ or d and $\hat{d}$ are on different sides of $\mathbf{S}$ then we mark the pixel as part of the PIP set.

The actual computation of $\hat{d}$ for a local point light source follows from trigonometry such that $\hat{d} = d \sin\beta / \sin(\alpha + \beta)$. Given the normal vector $\mathbf{n}$ and normalized light direction $\mathbf{L}$, then this is easily computed by using $\sin\beta = \mathbf{n} \cdot \mathbf{L}$. A similar calculation for directional lights is relatively straightforward. Note that is also possible to precompute Δ for the shadow map and store it for each pixel in addition to the depth value. This may be useful if the light source is mostly static since it reduces the memory bandwidth needed during the lighting pass of the rasterization algorithm.

One case that the method above does not detect is the geometric aliasing problem when a primitive is too small (e.g. a thin wire) as seen from the light view and thus not even drawn during scan-line rendering. Some of the pixels that are actually covered

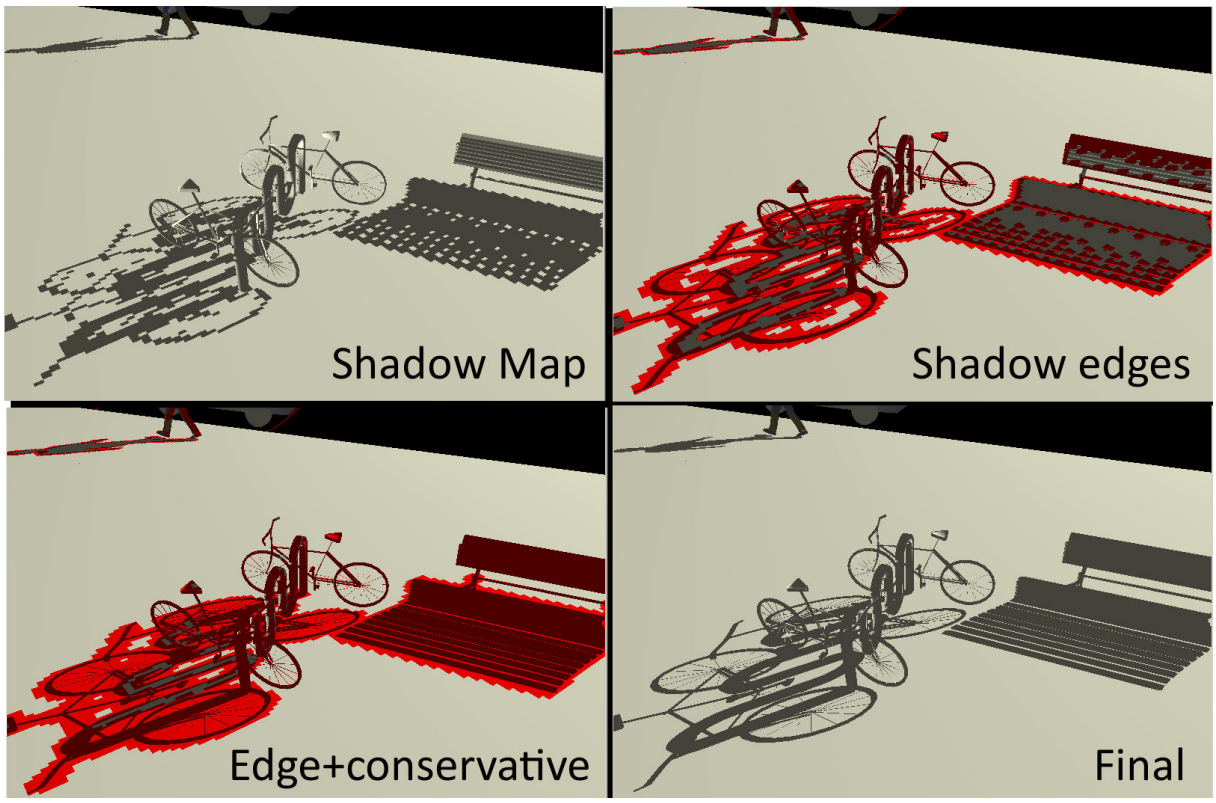


Figure 6.17: **Detecting shadow artifacts:** *Shadows on City model. Top left: shadows with perspective shadow mapping at* 2048^2 *resolution. Top Right: pixels marked for ray tracing in red. Bottom left: pixels marked by conservative rendering. Bottom right: final result. The image is identical to the fully ray-traced result.*

by the object may not get rasterized and thus not detected during edge filtering. This can cause missing shadow regions in the actual image. Our solution is to identify these small objects during rasterization from the light view and employ conservative rendering techniques to assure that they are actually part of the PIP set. We use geometry shaders to implement the conservative triangle rendering method described in [60] and modified by [134]. In essence, each triangle is transformed into a polygon with 6 corners by extruding at the original vertices. The new extruded primitive is then guaranteed to cover the center point of each pixel that it touches. Since extending all the triangles in the scene could be extremely costly, the shader also tests the area and aspect ratio of each triangle in image space and only uses conservative rendering for those that are thin and thus likely to be missed. At the same time, this technique automatically avoids very small but regular triangles (e.g. as in scanned models) where conservative rendering is not needed. Note that an even more efficient culling method would detect whether the triangle also has a silhouette edge, but we found that this adds more constraints on the rendering pipeline by having to provide adjacency information. Figure 6.17 illustrates the effect of our conservative rendering approach.

Soft shadows

We now describe an extension of the approach presented above that can also be used to render high quality soft shadows. Ray tracing approaches commonly stochastically generate a number of samples on the area light source for each hit point, evaluate their visibility using shadow rays and then average the results to find an estimate of how much of the light source is visible from the hit point. However, a high number of samples and shadow rays are needed to avoid high frequency aliasing error. We observe that the only area that needs to be evaluated from multiple light samples is the penumbra region, because for umbra and fully lit regions the visibility of the light is binary. This enables us to handle those binary regions using rasterization-based techniques, e.g. shadow

mapping, and to identify the penumbra as potentially incorrect pixels (PIP), and thereby perform selective ray tracing on these pixels. They only correspond to a portion of the final image, so significant amount of computation can be saved. More importantly, ray-based computation can be directed to regions that need it the most.

One standard shadow map taken from the center of the light suffices for our approach to estimate the penumbra region. For pixels in the image plane that do not fall in our estimated penumbra no shadow rays are needed and they will be shaded by shadow mapping. The penumbra is identified in the following manner. First we compute edge pixels in the shadow map in the same way we do for hard shadows. For hard shadows those edge pixels are the potentially inaccurate pixels, while for soft shadows they need to be expanded to account for the effect of the area light sources. Next we compute the projection of the area light onto the shadow map, centered at each of these silhouette pixels. This is equivalent to splatting each edge pixel using the shape of the light and a size based on the depth difference between the light and the edge. After splatting all the edge pixels we get a mask which is a union of all the splats in the map. Masked pixels are potentially in penumbra while unmasked pixels are either in the umbra or fully lit. We use this in a similar way as a shadow map. During rendering any points that are projected inside the masked area belong to the estimated penumbra and are ray traced using multiple shadow rays (e.g. 8×8 sampling), while points that are projected inside unmasked area are classified as umbra or fully lit based on the original shadow map. Figure 6.18 shows an example of this process.

The advantage of this approach is that it estimates the penumbra with the cost of little more than a standard shadow map, and consequently large parts of the image can be exempted from shadow ray tracing. Admittedly not all silhouette edges can be captured because the shadow map is generated only from the center of the light, but because such missed edges are often in the vicinity of the edges that are actually identified, it is very likely that most of the penumbra regions in the final image plane

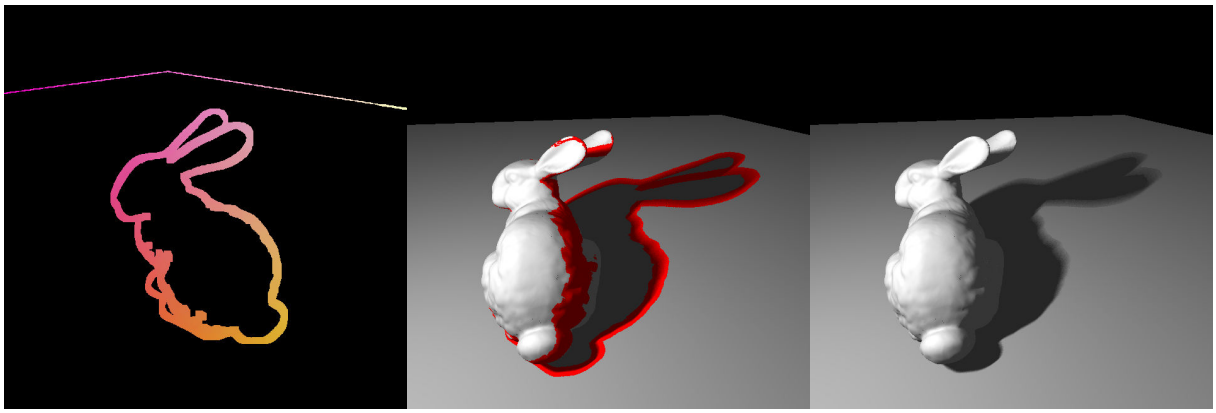


Figure 6.18: **Penumbra classification:** *Soft shadow on Bunny model. Left: mask in the shadow map formed by splatting edge pixels. Middle: penumbra pixels marked for ray tracing in red. Right: final result.*

have been correctly identified. The accuracy can be further improved by computing shadow maps from multiple samples on the area light source, e.g. the corners, and then computing the union of masked pixels from each.

6.3.4 Results

We have implemented the algorithms described above on a NVIDIA GPU. We use OpenGL with Cg as the rendering interface and CUDA for general-purpose programming. Our ray tracer uses a BVH built on the GPU as the acceleration structure with a stack-based traversal and persistent threads similar to those described in [2]. This also allows us to handle dynamic scenes by rebuilding the hierarchy each frame. A compaction step based on scan operations groups all the rays generated for marked pixels into a dense buffer that is then subdivided into small packets, each of which is handled independently. For rendering massive models, memory for storing the geometry and hierarchy on the GPU becomes an issue. We use a variant of the ReduceM representation shown in chapter 3, but we modify the strip representation slightly such that it is possible to also directly rasterize strips via OpenGL in order to use the same representation both for rasterization and ray tracing. Without this representation, larger models

Benchmarks	Tris	Hard			Soft		
		SM	SRT	FRT	SM	SRT	FRT
City	58K	256	103 (3%)	21	200	19 (9.8%)	3.7
Sibenik	82K	150	66 (4%)	13	47	7.3 (6%)	0.64
Buddha	1M	42	29 (1.4%)	8	34	9 (7.7%)	2.5
Powerplant	12M	25	16 (7.1%)	4	4.4	2.1 (11%)	0.8

Figure 6.19: **Performance:** *Performance of our selective ray tracing (SRT) approach on our benchmark models, compared to the light-space perspective shadow mapping algorithm (SM) with one point light for hard and 4 point lights for hard shadows, as well as a fully ray traced solution (FRT). The percentages show the fraction of pixels marked for ray tracing. All numbers are frames per second (FPS) at* 1024^2 *screen resolution.*

such as Powerplant would not fit into memory, especially not if stored in addition to a optimized vertex buffer representation for rasterizing geometry. In addition, we also use the top levels of the scene BVH for view frustum culling and occlusion culling based on occlusion queries both from the light as well as the camera view. These culling methods drastically accelerate the performance of the rasterization algorithm for large models. In our current implementation, view frustum culling is currently performed on the CPU, but could also be easily implemented in a CUDA algorithm. For soft shadows, we use a simple stratified sampling scheme for the shadow ray samples that is computed for each pixel with a simple random number generator inside the ray generation kernel. We base the random seed on sample location which allows us to compare our results for selective and full ray tracing without having to isolate variance in the estimate.

We now present results from our implementation running on a Intel Core2 Duo system at 2.83 GHz using a NVIDIA GTX 280 GPU running Windows XP x64. Fig. 6.19 summarizes the timings for hard and soft shadows at 1024×1024 screen resolution, including comparison timings for light-space perspective shadow mapping only, selective ray tracing and full ray tracing. The data for selective ray tracing also shows the percentage of pixels in the PIP set. We selected a wide range of benchmark models from relatively low-complexity game-like environments to high-complex scanned, CAD and architectural models to highlight the scalability of the algorithm. For shadows,

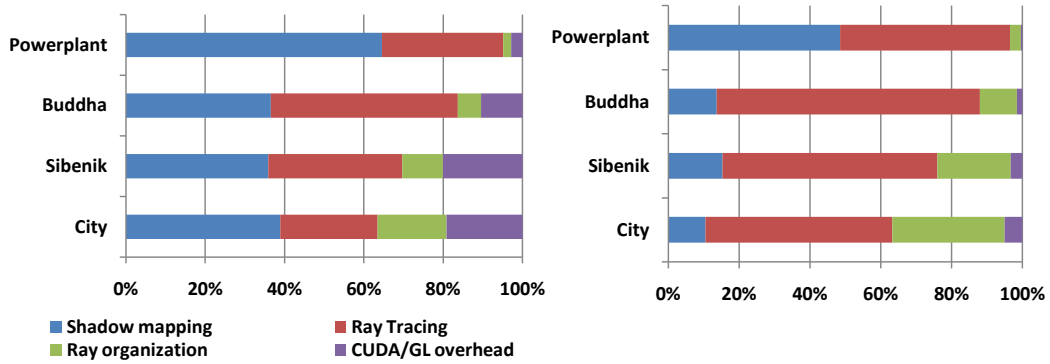


Figure 6.20: **Timings:** *Time spent in different parts of the algorithm for hard (left) and soft shadows (right). Rasterization includes both shadow map generation, frame buffer rendering and shading. Ray tracing includes time spent in the ray tracing kernel only, while ray generation is the time for generating the compact ray buffer from sparse frame buffer and writing back at the end. CUDA/GL times represent the cost for buffer transfers and similar as the overhead of switching between OpenGL and CUDA.*

all timings are for a moving light source, i.e. the shadow map is generated per frame, and soft shadows are generated by using 16 samples/pixel. Note that our algorithm is typically about 5 times faster than full ray tracing.

We present a detailed analysis of the timing breakdown in selective ray tracing (see Fig. 6.20) for several of our benchmark scenes. There is a relatively constant overhead associated with PIP detection, ray compaction and parts of the implementation such as the overhead of CUDA/OpenGL communication in the current programming environment. Note that the actual tracing of the selected ray samples only makes up a fraction of the time spent which explains why the render time does not decrease proportionally with the fraction of pixels ray traced. While we do not explicitly show the overhead in rasterization introduced by conservative rendering in the graph above, we have found that in practice it slows down the rasterization step by up to 10% (on Powerplant) since only a relatively small set of primitives are rendered conservatively.

6.3.5 Comparison

A very important aspect for evaluation of our algorithm is the difference in image quality compared to the full ray tracing solution. In practice, for hard shadows we have observed that our algorithm computes images that are almost error free and virtually identical to fully ray traced results for all our benchmark scenes, with differences arising from biasing errors. Unfortunately, we cannot guarantee full correctness since some features such as very small holes inside a solid object could theoretically be missed due to the regular sampling in light space (i.e. geometric aliasing errors). However, these artifacts appear to be very rare outside of specially constructed benchmark scenarios.

It is hard to directly compare the quality of our results with only rasterization-based approaches. The fast shadow mapping methods based on warping and partitioning may not account for projective aliasing [136, 161, 98] and their accuracy can vary based on the relative position of the light source w.r.t. the viewpoint. Many techniques to handle projective aliasing [96] and alias-free shadow maps [1, 75] can be implemented on current GPUs. These approaches can generate high quality shadows, but it is not clear whether they can scale well to massive models. For soft shadows, most of the accurate methods may not be able to handle complex models at interactive rates on current processors. Recently, Sintorn et al. [134] presented a soft shadow algorithm that can handle models with at most tens of thousands of triangles at interactive rates. It uses a more accurate method for penumbra computation and can be faster on smaller models, but quickly slows down on large models.

Our hybrid approach shares the same theme as other hybrid shadow generation algorithms that use shadow polygons and LODs [54] or shadow volumes [18]. However, LODs can affect the accuracy of the shadow boundaries. Moreover, the complexity of shadow volumes tense to increase with the number of silhouette edges in complex models. Thus, the bottleneck becomes the fill rate needed for rendering all volumes, which becomes prohibitively large with high geometric complexity. The analysis in [104] shows

that the overall expected number of silhouette edges for a model is proportional to the sum of the dihedral angles. As an example, the average dihedral angle per primitive on Powerplant model is about 6 times the average angle on the Buddha model. In our experiments with several viewpoints over 4 million silhouette edges may need to be rendered per frame for shadow volumes on Powerplant, which is clearly impractical. The idea of hybrid shadow mapping has been explored before. The method in [18] uses shadow mapping and employs a simple discontinuity detection on the depth map, then performs selective rasterization for the marked pixels using hierarchical z-culling. However, hybrid shadow volumes still have significant drawbacks compared to our approach. First, generating and processing the volumes including silhouette computation can be expensive especially for large CAD models with complex topology such the models used here. Second, even though shadow volumes may need to be selectively rasterized for only a small set of pixels, all shadow volumes still have to be processed by the rendering pipeline in any case. Hierarchical culling in the hardware may cull areas of the image efficiently, but the hybrid approach will most likely decrease the efficiency of the GPU rasterizer since active pixels will be relatively sparse and the parallel rasterization units are not sufficiently utilized by rendering only a few pixels inside a block of pixels.

Two approaches combining ray tracing and shadow mapping have been proposed: Beister *et al.* [10] perform CPU ray tracing for all pixels with shadow map discontinuities and also support soft shadows by rendering nine shadow maps and classifying regions as penumbra where the shadow map results disagree. However, this approach is not scalable particularly for large models since it requires rendering many shadow maps per frame (where rasterization may already be the bottleneck). In addition, in our experiments we found that using several shadow maps do not guarantee correct classification of penumbras in complex scenarios and may generate temporal aliasing artifacts (i.e. holes or blocks of shadows dropping in and out) which is particularly distracting. In addition, this approach may be limited by CPU-GPU communication. A subsequent approach

Model	Geometry	Full RT	SRT	SRT+SM	SRT % Rays
City	2 MB	1066 MB	113 MB	222 MB	5
Sibenik	2.2 MB	2280 MB	224 MB	859 MB	4
Buddha	27 MB	3148 MB	601 MB	1144 MB	12

Figure 6.21: **Memory bandwidth:** *Simulated memory bandwidth requirements for rendering one frame at* 512^2 *image resolution with selective (SRT) and full ray tracing (FRT) on several models (total storage for geometry and BVH is given in second column to show the total working set size.) The last column shows the time for SR plus a very conservative estimation of bandwidth needed by scan-line rendering of the shadow map.*

[66] performs ray tracing on the GPU, but uses a different method for detecting shadow pixels. While their method is also accurate for hard shadows, the overhead in their conservative rendering approach is much higher since they need to render all triangles as polygons. When directly comparing results, we observe that our ratio of ray traced pixels is significantly lower, i.e. our approach has to ray trace far less pixels. Even after adjusting for different hardware, it appears that our render times are substantially lower as a result. In relative performance, rendering accurate shadows in [66] is more than an order of magnitude slower than shadow mapping whereas we observe a factor of 1.5-2.5 in our results.

6.3.6 Performance analysis

In this section, we show that our hybrid algorithm maps well to current many-core GPUs and has lower memory bandwidth requirements as compared to full ray tracing. A key issue in designing GPGPU or related algorithms on current highly parallel architectures is to ensure that they are not limited by memory bandwidth. This is mainly because the growth rate for computational power far exceeds that of memory bandwidth. The streaming model of computation used in the rasterization pipeline has been shown to be very successful in this regard with memory bandwidth being mostly used for depth and frame buffer accesses.

We analyzed the memory bandwidth requirements when running both selective and

full ray tracing for two of our benchmark models, in particular the memory bandwidth used by the actual ray tracing kernel. Since current GPU architectures have limited cache sizes, i.e. only a texture cache, such analysis is simpler than for CPUs. We implemented a simple software simulator that emulates the behavior of the memory unit for global memory accesses in CUDA in device emulation mode running on the host CPU. Care has to be taken mainly to correctly account for the behavior of the memory unit in combining data parallel accesses to contiguous memory. Our results (see Fig. 6.21) indicate that selective ray tracing consistently only uses about an order of magnitude less memory bandwidth per frame than full ray tracing. The bandwidth for selective ray tracing is still slightly higher than expected from the number of rays compared to full ray tracing. This is due to the fact that the ray groups in selective ray tracing are more incoherent and thus will access more memory locations. In order to perform a complete analysis, we also need to compute the memory bandwidth needed by the rasterizer for scan-line rendering of the shadow map. However, this is not possible for hardware accelerated rasterization (i.e. current GPUs) since the implementation details are not publicly available. However, we provide an estimate for the bandwidth used for rasterizing the shadow map by multiplying the peak bandwidth on the GPU by the time taken for rasterization, thus providing us a conservative upper bound. We list the summed memory bandwidth for shadow mapping plus selective ray tracing in the last column of Fig. 6.21. Note that the combined bandwidth is still significantly lower than full ray tracing.

6.3.7 Scalability analysis

We also demonstrate the scalability of our approach with model complexity. To eliminate bias introduced by different model characteristics, we use different simplification levels of the same model and then compare the time used for selective ray tracing for each of the levels. Our results in Fig. 6.22 show that we can achieve sub-linear scalability

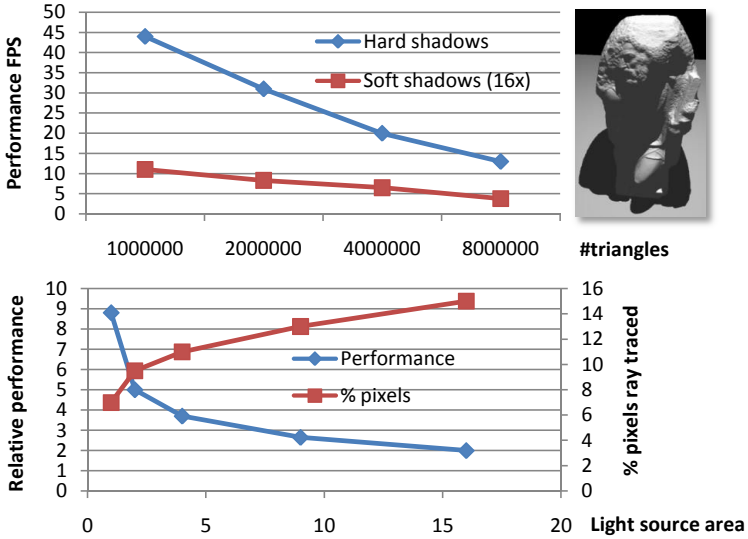


Figure 6.22: **Scalability:** *Top: Performance vs. model complexity on several simplification levels of the St. Matthew model, showing close to logarithmic scaling. Bottom: Performance vs. light source size on Buddha model.*

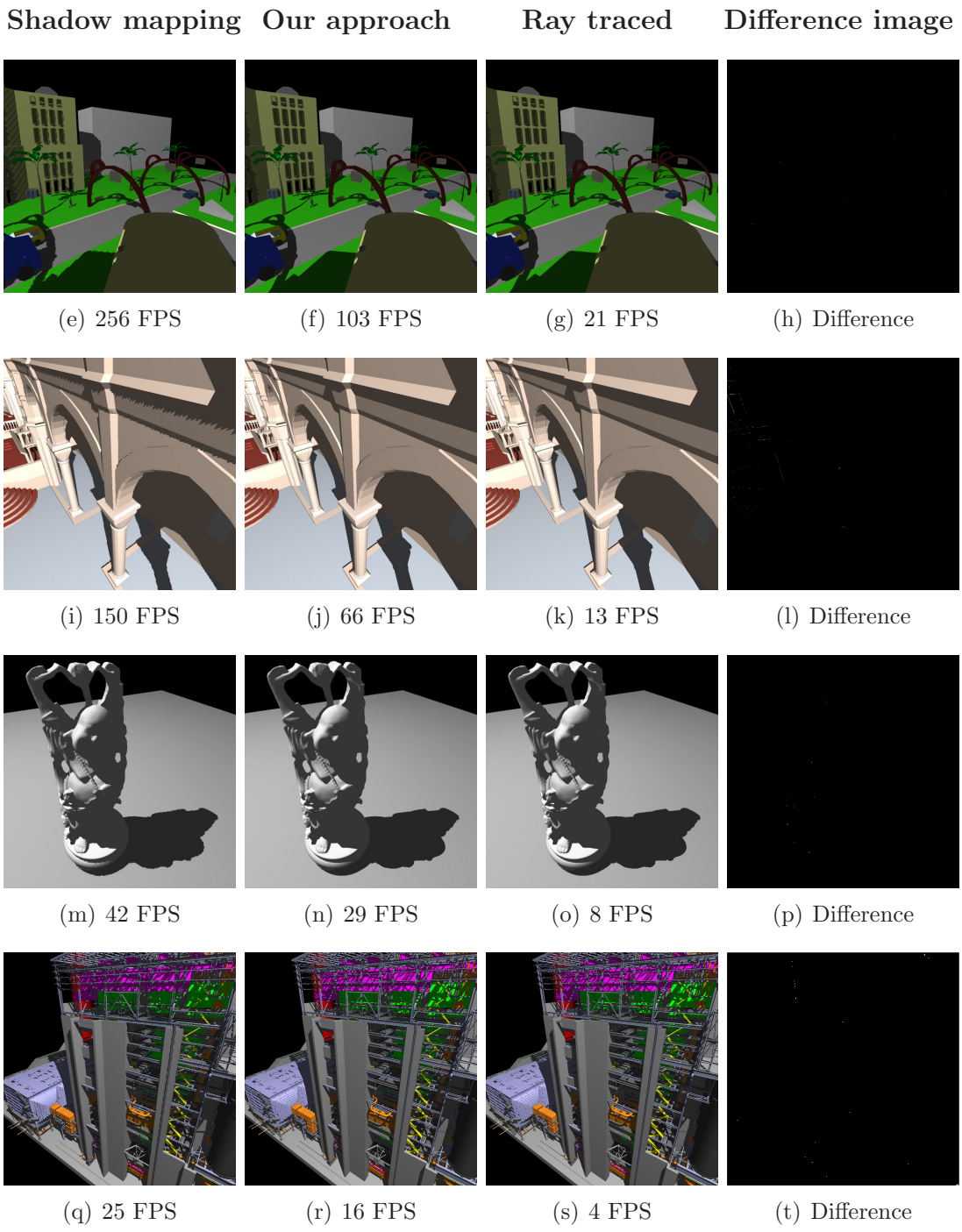
with model size due to the use of hierarchical structures and occlusion culling. We also look at the performance implications of increasing the area of the light source for soft shadow rendering (see Fig. 6.22 bottom). Similar to other approaches our performance decreases significantly with larger light sources mostly due to more of the pixels being in penumbra regions and subsequently being ray traced.

Overall, the main determining factor for our algorithm is the size of the PIP set. If the set is too large, then our algorithm cannot achieve a significant speedup over full ray tracing; however, at worst it can also only be slightly slower to the extent of shadow mapping overhead. In addition, the rays generated by selective ray tracing may exhibit less ray coherence than in full ray tracing which means that tracing these rays will be slightly more expensive on a per-ray metric. Since the rays still can access any part of the model, we also can only render models that fit into GPU memory and need to store an additional ray tracing hierarchy as well as update or rebuild it for deformable models. Our approach may also still carry over some of the geometric errors from rasterization such as depth buffer errors and resulting shadow map bias. In our soft shadow algorithm,

the main weakness is that penumbra radius in image space is necessarily conservative and can mark too many pixels for ray tracing in some cases. For example, very small objects that are extremely close to the light can result in a disproportionally large part of the depth buffer marked for ray tracing. In addition, the overall performance is linear too the number of shadow samples and more than 16 will most likely be required for high-quality rendering.

6.3.8 Conclusion

The results from the selective ray tracing approach show that is is possible to combine rasterization and ray tracing on the same GPU to combine the advantages of both approaches. As a combination of several methods presented in this thesis, it particularly is an application of the ReduceM massive model rendering approach and demonstrates that it works equally well on GPU architectures where the benefits of low memory overhead are even more important.



(e) 256 FPS (f) 103 FPS (g) 21 FPS (h) Difference

(i) 150 FPS (j) 66 FPS (k) 13 FPS (l) Difference

(m) 42 FPS (n) 29 FPS (o) 8 FPS (p) Difference

(q) 25 FPS (r) 16 FPS (s) 4 FPS (t) Difference

Figure 6.23: **Hard shadows:** *We compare the image fidelity of our algorithm to a pure ray-traced reference solution. From left to right: shadow mapping at* 1024^2*, selective ray tracing, ray traced reference, difference selective to reference.*

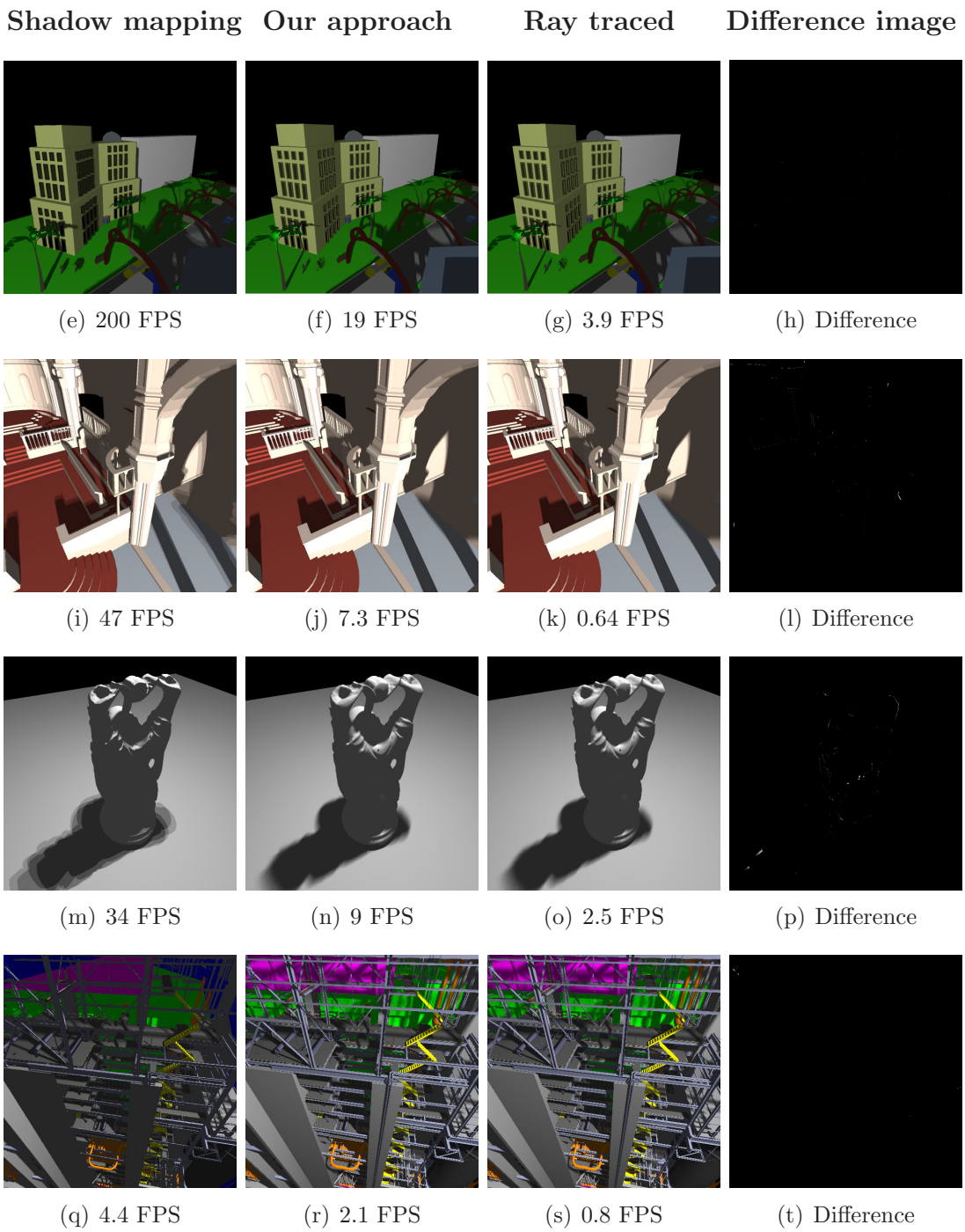


Figure 6.24: **Soft shadows:** *We compare the image fidelity of our algorithm to a pure ray-traced reference solution . From left to right: shadow mapping with 4 samples at* 1024^2*, selective ray tracing, ray traced reference (both at 16 samples/pixel), difference selective to reference.*

Chapter 7

Conclusion

In the previous chapters, we have introduced several new algorithms for interactive ray tracing of complex models to solve the memory and hierarchy maintenance issues presented in chapter 1. The problem of high memory footprint is addressed by the ReduceM representation that compactly represents triangle models for ray tracing and allows rendering of much larger models with limited memory with high performance. The hierarchy maintenance problem is addressed in two different ways: first, the deformable BVH algorithm can provide very fast hierarchy refitting to update an existing hierarchy between frames, and can also be implemented as a parallel algorithm. Second, the highly parallel construction algorithm can rebuild BVHs much quicker than a serial version could. Together, they provide a fast solution to general dynamic models for ray tracing.

Another aspect of this thesis was to show that these improvements for ray tracing are not limited to visualization purposes, but in fact can be used in many more applications. The sound simulation algorithm demonstrates the use of dynamic BVHs and data parallel ray packet techniques for high performance general geometric simulation, whereas the selective ray tracing approach uses the compact ReduceM representation on a GPU architecture to enable massive model rendering with strictly limited memory. This work has also shown that the parallelization techniques we developed for hierarchy

construction and refitting are useful in other hierarchy operations such as intersection of hierarchies for collision detection.

7.1 Future work

There are many avenues for future work to improve on the algorithms presented in this thesis. For compact model representations, the major remaining bottleneck is the representation of vertices and associated data, which is still uncompressed. To solve this problem, it would be interesting to investigate quantizing this data as well. This might also lead to a lossy compression scheme for ray tracing. In addition, the ReduceM representation is mostly orthogonal to many other techniques such as quantized bounding volumes, and it might be worthwhile to investigate possible improvements by combining them together.

Refitting techniques have been investigated thoroughly, but no algorithm that can fully cope with any kind of deformation or even insertion and deletion of objects has been proposed yet. Given the obvious time advantage compared to even the fastest construction methods, further improving refitting appears worthwhile. In particular, work into integrating localized rebuilding that so far still has high overhead might address many of these issues.

In general, the topic of parallel data structure construction and maintenance will most likely be of much further interest in the future, especially with the wide-spread use of highly-parallel architectures such as GPUs or the Larrabee processor [130] and more flexible rendering architectures. In this context, a possible direction for ray tracing includes the use of ray hierarchies in addition the scene hierarchies, both to improve coherence in ray intersection for wide vector units, as well as to improve the memory access pattern and working set size.

The future of work distribution on highly parallel architectures that is important

for a plethora of parallel algorithms is still highly dependent on the development of parallel hardware. At this time, synchronization support on these architectures is still rudimentary. However, future versions may shift the current state significantly and make other solutions much more efficient. Further investigation into best techniques on these processors will be necessary.

Bibliography

[1] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2004*, pages 161–166. Eurographics Association, 2004. 113, 125

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of High-Performance Graphics 2009*, 2009. 6, 20, 122

[3] V. Algazi, R. Duda, and D. Thompson. The CIPIC HRTF Database. In *IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics*, 2001. 91

[4] Thomas Annen, Zhao Dong, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Real-time, all-frequency shadows in dynamic scenes. *ACM Trans. Graph.*, 27(3):1–8, 2008. 114

[5] F. Antonacci, M. Foco, A. Sarti, and S. Tubaro. Real time modeling of acoustic propagation in complex environments. In *Proc. of 7th International Conference on Digital Audio Effects*, 2004. 81

[6] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968. 1, 17

[7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM. 100

[8] U. Assarsson and T. Akenine-Möller. A geometry-based soft shadow algorithm using graphics hardware. *ACM Transactions on Graphics*, 22(3):511–520, 2003. 114

[9] Louis Bavoil, Steven P. Callahan, and Claudio T. Silva. Robust soft shadow mapping with backprojection and depth peeling. *Journal of Graphics Tools*, 13(1), 2008. 114

[10] M. Beister, M. Ernst, and M. Stamminger. A hybrid gpu-cpu renderer. In *Proc. Vision, Modeling, and Visualization 2005*, pages 415–420, 2005. 113, 114, 126

[11] Carsten Benthin. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University, 2005. 37

[12] M. Bertram, E. Deines, J. Mohring, J. Jegorovs, and H. Hagen. Phonon tracing for auralization and visualization of sound. In *Proceedings of IEEE Visualization 2005*, pages 151–158, 2005. 80

[13] J. Borish. Extension of the image model to arbitrary polyhedra. *Journal of the Acoustical Society of America*, 75(6):1827–1836, 1984. 81

[14] G. Bradshaw and C. O'Sullivan. Adaptive medial-axis approximation for sphere-tree construction. *ACM Trans. on Graphics*, 23(1), 2004. 99

[15] Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. Out-of-core data management for path tracing on hybrid resources. *Comput. Graph. Forum*, 28(2):385–396, 2009. 7, 21

[16] Loren Carpenter. The a-buffer, an antialiased hidden surface method. In *Proc. of ACM SIGGRAPH*, pages 103–108, 1984. 87

[17] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The Ray Engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. 19

[18] Eric Chan and Frédo Durand. An efficient hybrid shadow rendering algorithm. In *Proceedings of the Eurographics Symposium on Rendering*, pages 185–195. Eurographics Association, 2004. 113, 125, 126

[19] Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. Parallel sah k-d tree construction. In *Proc. High Performance Graphics 2010*, 2010. 27

[20] Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum*, 22(3):543–552, September 2003. 7, 22

[21] David Cline, Kevin Steele, and Parris K. Egbert. Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools: JGT*, 2006. 8, 24, 47

[22] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, 1984. 17

[23] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH '77)*, pages 242–248, 1977. 113

[24] S. Curtis, R. Tamstorf, and D. Manocha. Fast collision detection for deformable models using representative-triangles. *Proc. of ACM Symposium on Interactive 3D Graphics and Games*, 2008. 99, 111

[25] Bengt-Inge Dalenbck, Peter Svensson, and Mendel Kleiner. Room acoustic prediction and auralization based on an extended image source model. *The Journal of the Acoustical Society of America*, 92(4):2346, 1992. 81

[26] E. Deines, M. Bertram, J. Mohring, J. Jegorovs, F. Michel, H. Hagen, and G.M. Nielson. Comparative visualization for wave-based and geometric acoustics. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), 2006. 80

[27] David E. DeMarle, Christiaan P. Gribble, Solomon Boulos, and Steven G. Parker. Memory sharing for interactive ray tracing on clusters. *Parallel Comput.*, 31(2):221–242, 2005. 21

[28] David E. DeMarle, Christiaan P. Gribble, and Steven G. Parker. Memory-savvy distributed interactive ray tracing. In *EGPGV*, pages 93–100, 2004. 7

[29] A. Dietrich, A. Stephens, and I. Wald. Exploring a boeing 777: Ray tracing large-scale cad data. *IEEE Computer Graphics and Applications*, 27(6):36–46, 2007. 21

[30] M. Dillencourt. Finding hamiltonian cycles in delaunay triangulartions is np-complete. *Canadian Conference on Computational Geometry*, 1992. 40

[31] Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. Razor: An architecture for dynamic multiresolution ray tracing. In *Conditionally accepted to ACM Transactions on Graphics*, 2007. 7, 22, 27

[32] I. A. Drumm. *The Development and Application of an Adaptive Beam Tracing Algorithm to Predict the Acoustics of Auditoria*. PhD thesis, 1997. 81, 97

[33] Stephen A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *of Eurographics2001*, pages 500–510, 2001. 104

[34] M. Eisemann, T. Grosch, M. Magnor, and S. Mueller. Automatic Creation of Object Hierarchies for Ray Tracing Dynamic Scenes. In Vaclav Skala, editor, *WSCG Short Papers Post-Conference Proceedings*. WSCG, 1 2007. 28

[35] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2004. 99

[36] Jacob Munkberg Erik Mansson and Tomas Akenine-Moller. Deep coherent ray tracing. *Symp. on Interactive Ray Tracing*, 2007. 7, 22

[37] Manfred Ernst and Gnther Greiner. Early split clipping for bounding volume hierarchies. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 2007. 75

[38] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization*, pages 319–326, 1996. 40, 43, 45

[39] A. Farina. Ramsete - a new pyramid tracer for medium and large scale acoustic problems. In *Proceedings of EURO-NOISE*, 1995. 81

[40] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proc. ACM SIGGRAPH/EG Conf. on Graphics Hardware*, pages 15–22, 2005. 20, 114

[41] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998. 100

[42] T. Funkhouser, N. Tsingos, I. Carlbom, G. Elko, M. Sondhi, J. West, G. Pingali, P. Min, and A. Ngan. A beam tracing method for interactive architectural acoustics. *Journal of the Acoustical Society of America*, 115(2):739–756, February 2004. 81, 87

[43] Thomas Funkhouser, Ingrid Carlbom, Gary Elko, Gopal Pingali, Mohan Sondhi, and Jim West. A beam tracing approach to acoustic modeling for interactive virtual environments. In *Proc. of ACM SIGGRAPH*, pages 21–32, 1998. 81

[44] Thomas Funkhouser, Nicolas Tsingos, and Jean-Marc Jot. Survey of methods for modeling sound propagation in interactive virtual environment systems. *Presence and Teleoperation*, 2003. 90

[45] Thomas A. Funkhouser, Patrick Min, and Ingrid Carlbom. Real-time acoustic modeling for distributed virtual environments. In *Proc. of ACM SIGGRAPH*, pages 365–374, 1999. 81

[46] Kirill Garanzha. Efficient clustered bvh update algorithm for highly-dynamic models. In *Proc. IEEE Symposium on Interactive Ray Tracing*, pages 123–130, 2008. 28

[47] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum*, 29(2), May 2010. 22

[48] Jon Genetti and Dan Gordon. Ray tracing with adaptive supersampling in object space. In *Graphics Interface '93*, pages 70–77, 1993. 97

[49] Andrew S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988. 9, 24

[50] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987. 25, 38, 52, 66

[51] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, pages 171–180, 1996. 50, 99, 105, 107

[52] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. *Proc. of ACM SIGMOD*, 2006. 76

[53] N. Govindaraju, D. Knott, N. Jain, I. Kabal, R. Tamstorf, R. Gayle, M. Lin, and D. Manocha. Collision detection between deformable models using chromatic decomposition. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)*, 24(3):991–999, 2005. 99, 112

[54] N. Govindaraju, B. Lloyd, S. Yoon, A. Sud, and D. Manocha. Interactive shadow generation in complex environments. *Proc. of ACM SIGGRAPH/ACM Trans. on Graphics*, 22(3):501–510, 2003. 125

[55] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–32, 2003. 99, 112

[56] I. Grinberg and Y. Wiseman. Scalable parallel collision detection simulation. *Proc. of Signal and Image Processing*, 2007. 100

[57] Johannes Günther, Heiko Friedrich, Ingo Wald, Hans-Peter Seidel, and Philipp Slusallek. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3), September 2006. 9, 24

[58] Johannes Gnther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Real-time Ray Tracing on GPU with BVH-based Packet Traversal. In *Proc. IEEE/EG Symposium on Interactive Ray Tracing*, pages 113–118, 2007. 74

[59] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. *Computer Graphics Forum*, 22(4):753–774, dec 2003. 113

[60] Jon Hasselgren, Tomas Akenine-Mller, and Lennart Ohlsson. Conservative rasterization on the gpu. *GPU Gems 2*, pages 677–690, 2005. 120

[61] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 2, 3, 8, 19, 25, 38, 51, 52, 66

[62] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On Fast Construction of Spatial Hierarchies for Ray Tracing. *Submitted to RT'06*, 2006. 61

[63] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Proc. of ACM SIGGRAPH*, pages 119–127, 1984. 81

[64] B. Heidelberger, M. Teschner, and M. Gross. Real-time volumetric intersections of deforming objects. *Proc. of Vision, Modeling and Visualization*, pages 461–468, 2003. 99, 112

[65] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distrib. Comput.*, 18(3):189–207, 2006. 100

[66] S. Hertel, K. Hormann, and R. Westermann. A hybrid GPU rendering pipeline for alias-free hard shadows. In D. Ebert and J. Krüger, editors, *Eurographics 2009 Areas Papers*, pages 59–66, München, Germany, March/April 2009. Eurographics Association. 113, 114, 127

[67] H. Hoppe. Optimization of mesh locality for transparent vertex caching. *Proc. of ACM SIGGRAPH*, pages 269–276, 1999. 38

[68] Daniel Horn. Stream reduction operations for gpgpu applications. *GPU Gems 2*, 2005. 68

[69] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In *Proc. I3D '07*, pages 167–174, 2007. 20, 114

[70] P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993. 50

[71] Warren Hunt, William R. Mark, and Gordon Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, September 2006. 26, 69, 75

[72] Homan Igehy. Tracing ray differentials. In *ACM SIGGRAPH*, pages 179–186, 1999. 22

[73] T. Jensen and B. Toft. *Graph Coloring Problems*. Wiley InterScience, 1995. 1

[74] Gregory S. Johnson, Warren A. Hunt, Allen Hux, William R. Mark, Christopher A. Burns, and Stephen Junkins. Soft irregular shadow mapping: fast, high-quality, and robust soft shadows. In *Proc. I3D '09*, pages 57–66, 2009. 114

[75] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*, 24(4):1462–1482, 2005. 113, 125

[76] Dan Gordon Jon Genetti and Glen Williams. Adaptive supersampling in object space using pyramidal rays. *Computer Graphics Forum*, 17(1):29–54, 1998. 97

[77] C. Joslin and N. Magnetat-Thalmann. Significant facet retrieval for real-time 3d sound rendering. In *Proceedings of the ACM VRST*, 2003. 84

[78] James T. Kajiya. The rendering equation. In *Proc. of ACM SIGGRAPH*, pages 143–150, 1986. 1

[79] B. Kapralos, M. Jenkin, and E. Milios. Acoustic modeling utilizing an acoustic version of phonon mapping. In *Proc. of IEEE Workshop on HAVE*, 2004. 80

[80] D. Kim, J. Heo, and S. Yoon. PCCD: Parallel continuous collision detection. Technical report, http://sglab.kaist.ac.kr/PCCD/, Korea Advanced Institute of Science and Technology, South Korea, 2008. 99, 111

[81] Duk-Su Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung eui Yoon. HPCCD: hybrid parallel continuous collision detection. In *Computer Graphics Forum (Proc. Pacific Graphics, to appear)*, 2009. 99

[82] Tae-Joon Kim, Bochang Moon, Duksu Kim, and Sung-Eui Yoon. RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 2009. 8

[83] Y. Kitamura, A. Smith, H. Takemura, and F. Kishino. Parallel algorithms for real-time colliding face detection. *Robot and Human Communication, 1995. RO-MAN'95 TOKYO, Proceedings., 4th IEEE International Workshop on*, pages 211–218, Jul 1995. 100

[84] J. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics*, 4(1):21–37, 1998. 50, 99, 104, 105

[85] D. Knott and D. K. Pai. CInDeR: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface*, pages 73–80, 2003. 99, 112

[86] A. Krokstad, S. Strom, and S. Sorsdal. Calculating the acoustical room response by the use of a ray tracing technique. *Journal of Sound and Vibration*, 8(1):118–125, July 1968. 80

[87] Vipin Kumar and Ananth Y. Grama. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22:60–79, 1994. 100

[88] K. Kunz and R. Luebbers. *The Finite Difference Time Domain for Electromagnetics*. CRC Press, 1993. 80

[89] K. H. Kuttruff. Auralization of impulse responses modeled on the basis of ray-tracing results. *Journal of Audio Engineering Society*, 41(11):876–880, November 1993. 1, 80

[90] Samuli Laine. *Efficient Physically-Based Shadow Algorithms*. PhD thesis, Helsinki University of Technology, 2006. 113

[91] Samuli Laine. Restart trail for stackless BVH traversal. In *Proceedings of High-Performance Graphics 2010*, 2010. 20

[92] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Distance queries with rectangular swept sphere volumes. *Proc. of IEEE Int. Conference on Robotics and Automation*, pages 3719–3726, 2000. 99, 104, 105

[93] C. Lauterbach, S.i Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. *IEEE Symposium on Interactive Ray Tracing*, 2006. 25, 45, 74, 85

[94] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Proc. Eurographics '09*, 2009. 27, 72, 100

[95] Scott Le Grand. Broad-phase collision detection with cuda. August 2007. 99

[96] Aaron E. Lefohn, Shubhabrata Sengupta, and John D. Owens. Resolution-matched shadow maps. *ACM Trans. Graph.*, 26(4):20, 2007. 113, 125

[97] Hilmar Lehnert. Systematic errors of the ray-tracing algorithm. *J. Applied Acoustics*, 38(2-4):207–221, 1993. 80

[98] Brandon Lloyd. *Logarithmic Perspective Shadow Maps*. PhD thesis, University of North Carolina at Chapel Hill, 2007. 113, 125

[99] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002. 22

[100] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 1990. 3, 25, 38, 66

[101] Jeffrey Mahovsky. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, September 2005. 8, 23, 47, 52, 57, 58

[102] Jeffrey Mahovsky and Brian Wyvill. Fast ray-axis aligned bounding box overlap tests with plcker coordinates. *journal of graphics tools*, 9(1):35–46, 2004. 51

[103] Michael D. McCool. Shadow volume reconstruction from depth maps. *ACM Trans. Graph.*, 19(1):1–26, 2000. 113

[104] Morgan McGuire. Observations on silhouette sizes. *jgt*, 9(1):1–12, 2004. 125

[105] Qi Mo, Voicu Popescu, and Chris Wyman. The soft shadow occlusion camera. *Proc. Pacific Graphics 2007*, pages 189–198, 2007. 114

[106] T. Moller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2), 1997. 108

[107] Bochang Moon, Youngyong Byun, Tae-Joon Kim, Pio Claudio, and Sung-Eui Yoon. Cache-oblivious ray reordering. Technical Report CS-TR-2009-314, 2009. 21

[108] Koji Nakamaru and Yoshio Ohno. Enhanced breadth-first ray tracing. *journal of graphics, gpu, and game tools*, 6(4):13–28, 2001. 21

[109] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. 73

[110] B. Ooi, K.J. McDonell, and R. Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. *Proc. of the IEEE COMPSAC Conf.*, 1987. 32

[111] T. Otsuru, Y. Uchinoura, R. Tomiku, N. Okamoto, and Y. Takahashi. Basic concept, accuracy and application of large-scale finite element sound field analysis of rooms. In *Proc. ICA 2004 (Kyoto)*, pages I–479–I–482, April 2004. 80

[112] Jacopo Pantaleoni and David Luebke. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. High Performance Graphics 2010*, 2010. 27

[113] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):1–13, 2010. 20

[114] Steven G. Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian E. Smits, and Charles D. Hansen. Interactive ray tracing. In *SI3D*, pages 119–126, 1999. 21

[115] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proc. of ACM SIGGRAPH*, pages 101–108, 1997. 7, 21

[116] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94, September 2006. 26, 69, 72

[117] Stefan Popov, Johannes Gnther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum (Proc. EUROGRAPHICS)*, 26(3):415–424, 2007. 19, 20

[118] X. Provot. Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface*, pages 177–189, 1997. 108

[119] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM. 19

[120] A. Rajkumar, B. F. Naylor, F. Feisullin, and L. Rogers. Predicting rf coverage in large environments using ray-beam tracing and partitioning tree represented geometry. *Wirel. Netw.*, 2(2):143–154, 1996. 97

[121] V. Nageshwara Rao and Vipin Kumar. Parallel depth-first search, part i: Implementation. *International Journal of Parallel Programming*, 16:6–479, 1987. 100

[122] A. Reshetov. Faster ray packets - triangle intersection through vertex culling. *Symp. on Interactive Ray Tracing*, 2007. 48

[123] Alexander Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. *Symposium on Interactive Ray Tracing*, 0:57–60, 2006. 64

[124] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005. 18, 58, 62, 64, 84, 91

[125] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. In *Proc. Eurographics Symposium on Rendering*, pages 99–110, June 2007. 22

[126] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980. 25

[127] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), August 2007. 45, 46

[128] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *IPDPS*, pages 1–10. IEEE, 2009. 68, 72

[129] Michael Schwarz and Marc Stamminger. Bitmask soft shadows. *Comput. Graph. Forum*, 26(3):515–524, 2007. 114

[130] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008. 134

[131] Maxim Shevtsov, Alexie Soupikov, and Alexander Kapustin. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum*, 26(3):395–404, September 2007. 26

[132] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. AK Peters Limited, second edition, 2003. 51

[133] Ken Shoemake. Pluecker coordinate tutorial. *Ray Tracing News*, 11(1), 1998. 35, 86

[134] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample-based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proc. EGSR '07)*, 27(4):1285–1292, 2008. 114, 120, 125

[135] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools: JGT*, 3(2):1–14, 1998. 19

[136] Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proc. SIGGRAPH '02*, pages 557–562, 2002. 113, 116, 125

[137] Marc Stamminger, Jrg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective textures. *Proc. Eurographics Workshop on Rendering '00*, pages 377–388, 2000. 114

[138] A. Stephens, S. Boulos, J. Bigler, I. Wald, and S. Parker. An application of scalable massive model interaction using shared memory systems. *Proc. of Eurographics Symp. on Parallel Graphics and Visualization*, pages 19–26, 2006. 21, 60

[139] U. Stephenson. Quantized pyramidal beam tracing - a new algorithm for room acoustics and noise immission prognosis. *Acustica - Acta Acustica*, 82(3):517–525, 1996. 97

[140] A. Sud, N. Govindaraju, R. Gayle, I. Kabul, and D. Manocha. Fast proximity computation among deformable models using discrete voronoi diagrams. *Proc. of ACM SIGGRAPH*, pages 1144–1153, 2006. 99, 112

[141] Min Tang, Sean Curtis, Sung-Eui Yoon, and Dinesh Manocha. Interactive continuous collision detection between deformable models using connectivity-based culling. In *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 25–36, New York, NY, USA, 2008. ACM. 99

[142] N. Thrane and Lars-Ole Simonsen. *A comparison of acceleration structures for GPU assisted ray tracing*. Master's thesis, University of Aarhus, Aarhus, Denmark, 2005. 19

[143] R. Tomiku, T. Otsuru, Y. Takahashi, and D. Azuma. A computational investigation on measurements in reverberation rooms by finite element sound field analysis. In *Proc. ICA 2004 (Kyoto)*, pages II–941–II–942, April 2004. 80

[144] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–14, 1997. 99

[145] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007. 41, 45, 85

[146] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). SCI Institute Technical Report UUSCI-2006-009, University of Utah, 2006. 52, 61

[147] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proc. of IEEE Symp. on Interactive Ray Tracing*, pages 61–69, 2006. 66

[148] I. Wald, W. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S. Parker, and P. Shirley. State of the art in ray tracing dynamic scenes. *Eurographics State of the Art Reports*, 2007. 13

[149] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 6, 11, 25, 45, 51

[150] Ingo Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007. 26, 72, 73, 74, 75

[151] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (EUROGRAPHICS)*, volume 20, pages 153–164, 2001. 3, 17, 51, 58

[152] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007. 25, 28, 57, 74

[153] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proc. of the Eurographics Symp. on Rendering*, 2004. 8, 23, 45, 58

[154] Ingo Wald, Christiaan P Gribble, Solomon Boulos, and Andrew Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, 2007. 22

[155] Ingo Wald, Vlastimil Havran, Ingo Wald, and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–70, 2006. 25

[156] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In Dieter Schmalstieg and Jiří Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116. The Eurographics Association, September 2007. 6, 24

[157] L. M. Wang, J. Rathsam, and S. R. Ryherd. Interactions of model detail level and scattering coefficients in room acoustic computer simulation. In *International Symposium on Room Acoustics: Design and Science*, 2004. 97

[158] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980. 1, 17, 97

[159] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools: JGT*, 10(1):49–54, 2005. 51

[160] L. Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274, 1978. 113

[161] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Proc. of the Eurographics Symposium on Rendering*, pages 143–152, 2004. 113, 125

[162] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, 2006. 62, 63

[163] Carsten Wchter and Andreas Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006: Eurographics Symposium on Rendering.*, 2006. 25, 26, 27, 32, 34, 61, 62, 64, 72, 74

[164] S. Yoon, C. Lauterbach, and D. Manocha. R-LODs: Interactive LOD-based Ray Tracing of Massive Models. *The Visual Computer (Pacific Graphics)*, 2006. 7, 23

[165] Sungeui Yoon, Sean Curtis, and Dinesh Manocha. Ray tracing dynamic scenes using selective restructuring. *Proc. of Eurographics Symposium on Rendering*, 2007. 28

[166] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *Proc. SIGGRAPH Asia*, 2008. 27, 78, 100